UNIX Programming Guide – Part 3

Content Summary

Directories in UNIX

Introduction

UNIX systems implement directories also as “regular files”, except that there is a special distinguishing bit set in their inode that differentiates the directory from regular files. The kernel does not permit writing on that bit.

Directories render the hierarchical file structure to UNIX. They are instrumental in mapping a file name to an inode number. Recall that an inode number is a unique number associated with every file. The inode number indexes to a data structure called inode that holds information about a file.

A directory can be formally defined as a regular file whose data is a sequence of entries, each consisting of an inode number and the name of the file contained in that directory. Every directory contains the file names “.” and “..” whose inode numbers are those of the directory and the parent directory. In UNIX systems, the file system hierarchy starts with the “root” directory and hence the kernel loads the inode values of “.” and “..” in the “root” directory during file system initialization.

Special system calls can manipulate directories but the kernel reserves the exclusive right to write to a directory so as to ensure the correct structure of a directory.

Search Paths and Current Working Directory

Assume you have created an executable file called “ls” that resides in your current working directory (which is by default your “home” directory). Also note that another executable file called “ls” would exist in your system (most likely in the path “/bin/ls”). Now when you type “ls”, which of the two executables would get invoked?

The answer depends on the value of the PATH variable.
Whenever UNIX encounters an executable it searches for it in systematic way – using the PATH variable. The PATH variable is one of the constituents of a collection of variables called environment variables (More about environment variables later). A typical value of the PATH variable would like below:

```
/usr/bin:/etc:/usr/local/bin/:usr/ccs/bin:/home/myname/bin:.
```

PATH contains the fully qualified pathnames of important directories separated by colons. This means that when UNIX encounters an executable it would first search in "`/usr/bin`". If not found then it would look in "`/etc`" and then in "`/usr/local/bin`" and so on. If the executable is not located even after traversing all the paths listed in the PATH variable then an error message is flashed.

To see the actual value of PATH variable in your machine, type

```
> echo $PATH
```

getcwd System Call

To get the current working directory in UNIX, we use the "pwd" command. The system call behind the "pwd" command is the getcwd() call.

```
# include<unistd.h>
char *getcwd(
char *buf, /* Returned pathname */
size_t bufsize /* sizeof buf */ ???
);
/* Returns pointer to 'buf' on success and NULL on
error */
```

The getcwd() call copies the current working directory path into 'buf'. Note that 'buf' cannot be null. The second argument denotes the size of the buffer. Ideally the size of the buffer should be the maximum length of a path. To be able to find the maximum path length in a UNIX system, we use the pathconf() call. The call below finds the maximum path length starting from the "root"

```
long max= pathconf("/",_PC_PATH_MAX);
```

A program that uses the getcwd() would look like below:

```
# include<stdio.h>
# include<unistd.h>
int main(void) {
long max;
char *buf;
```

```
max= pathconf("/",_PC_PATH_MAX);
buf=(char*)malloc(max);
getcwd(buf,max);
printf("%s\n",buf);

return 0;
}
```

chdir System Call

The command to change directory in UNIX is the cd command. The system call behind the cd command is the chdir system call.

```
# include<unistd.h>
int chdir(
      const char *path
); /* Returns zero on success and -1 on error */
```

This system call changes the current working directory to that specified in "path". The "path" argument can be a relative path or the absolute path. Whatever inode that path leads to (if it's a directory inode) becomes the current working directory.

Making and Removing Directories

mkdir System Call

```
# include<sys/stat.h>
int mkdir(
      const char *path,      /* Pathname */
      mode_t perms           /* Permissions */
 );
/* Returns 0 on success and -1 on error */
```

The semantics of this call is very similar to open() call. The mkdir() call automatically creates the "." and ".." links.

rmdir System Call

```
# include<unistd.h>
int rmdir(
      const char *path       /* Pathname */
 );
/* Returns 0 on success and -1 on error */
```

Every directory in UNIX is nested inside another directory (except for the root directory). When removing a directory, the link in the parent directory that leads to the directory to be deleted will be removed. On removing the link, the link count of the corresponding inode(the deleted directory) reduces by one. If the link count becomes zero, the file system will discard that inode.

Strictly, removing a directory only means removing a link to the directory.

A restriction on removing directories is that the directory to be removed has to be empty. If it is not empty, you have to remove the files in directory, other directories (and in turn the contents of those directories) and so on. In fact, the command "rm –r", recursively deletes a directory, as described above.

Opening, Reading and Closing Directories

Given that directories are also regular files, it would be natural for open() and read() calls to work on directories as well. In fact, in FreeBSD and Solaris systems, trying to open and read directories using the standard read() and open() calls, would give some difficult-to-read output . But these generic calls do not work on modern linux systems. There are standardized system calls specifically for directory manipulations and they are described below.

opendir and closedir System Calls

```
# include<dirent.h>
DIR* opendir(
        const char* path /* directory pathname */
        );
/* Returns a DIR pointer or NULL on error */
```

The opendir() call returns a pointer to a directory stream (much like how the standard C function fopen() would return a pointer to a FILE type). This DIR pointer is used as an argument in other directory manipulation functions. The returned pointer points to the first entry in the directory.

```
# include<dirent.h>
int closedir(
        DIR *dirp /*DIR pointer from opendir */
        );
/* Returns a 0 on success or -1 on error */
```

closedir() is just an explicit way of informing the kernel that you are done with the use of DIR.

readdir System Call and *dirent* structure

```
#include <dirent.h>
struct dirent *readdir(
                  DIR *dirp  /* DIR Pointer from opendir */
);
/* Returns structure or NULL on EOF or error */
```

readdir () returns a pointer to the dirent structure which contains a i-number and name as shown below

```
struct dirent {
      ino_t d_ino;     /* i-number */
      char d_name[];   /* name */
};
```

The i-number and name returned would correspond to single entry. Hence to read all entries in a directory, we would call readdir in a loop until EOF is reached.

Program that mimics the "ls" command

```
# include<dirent.h>
# include<stdio.h>

int main(){
      struct dirent *direntp;
      DIR *dirp;

      dirp=opendir("."); /* Open the current directory */
      while((direntp = readdir(dirp)) != NULL){
            printf("%s \n",direntp->d_name);
      }
      closedir(dirp);
      return 0;
}
```

Note that the output of the program is not exactly the same as that of "ls" command. The output of the program is not sorted. In fact, there is no control over the order in which entries are actually stored or read from the directory.

Scanning a directory

Scanning a directory is looking into the contents of the directory. A more sophisticated function to perform directory scans is called the scandir() call.

scandir System Call

```
#include <dirent.h>
int scandir(
       const char *dir,            /* Directory to be scanned */
       struct dirent ***namelist,
       int (*select)(const struct dirent *),
       int (*compar)(const struct dirent **, const struct dirent **)
       );
```

The scandir() function scans the directory *dir*, calling *filter()* on each directory entry. Entries for which *filter()* returns nonzero are stored in strings allocated via malloc(). These strings are sorted using the comparison function *compar()*, and then collected in array *namelist* which is allocated via malloc(). If *filter()* is NULL, all entries are selected.

One of the commonly used *compar()* functions is the inbuilt *alphasort()* function, given below

```
int alphasort(const struct dirent **a, const struct dirent **b);
```

The following example clarifies the above explanation.

Program to print the contents of a directory in reverse sorted order

```
#include <dirent.h>
#include <stdio.h>
int main(void){
      struct dirent **namelist;
      int n;

      n = scandir(".", &namelist, NULL, alphasort);
      if (n < 0)
            printf("scandir error\n");
      else {
            while(n--) {
              printf("%s\n", namelist[n]->d_name);
             }
      }
return 0;
}
```

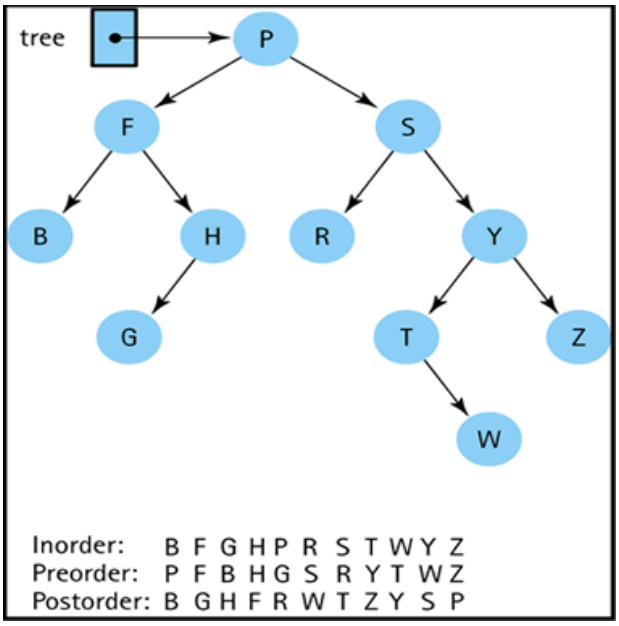
In this program, the *filter()* function is not used. Hence all entries in the current directory (“.”) would be returned. Each of these entries would be sorted alphabetically (using *alphasort()* function) and allotted to *namelist*. The while loop just prints the contents of the *namelist*.

Program to print only the c program files in a directory

```
 #include <dirent.h>
#include <stdio.h>
#include <string.h>

int filter(struct dirent *entry);

int main(void){
    struct dirent **namelist;
    int n;

    n = scandir(".", &namelist, filter , alphasort);
    if (n < 0)
        printf("scandir error");
    else {
        while(n--) {
            printf("%s\n", namelist[n]->d_name);
        }
    }
return 0;
}

int filter(struct dirent *entry)
{
char *name=entry->d_name;
if(name[strlen(name)-1]=='c' && name[strlen(name)-2]=='.')
     return 1;
else
     return 0;
```

Exercises

1. Write a program that works like the “rm –r” command. The program should scan a directory recursively and delete all the contents.
2. Interactive File Deletion – Pick out only the “.txt” files from the current working directory. For files, whose size exceeds 1KB, ask the user if he wants to delete the file. If affirmative, delete that particular file. After all the deletions, display the contents of the directory.
3. Write a program that will calculate the size of a given directory. To be able to do this, perform a post-order traversal on the unix directory structure starting from the given directory. While traversing each node, sum up the size. The size of a directory is the sum of all its children (other directories and files).



In the above tree, to calculate of the size of S, we first need to find the sum of sizes of R and Y. To find size of Y, we calculate the sum of sizes of T and Z. And size of T needs the size of W.
Hence a postorder traversal would be : RWTZYS