UNIX System Programming
Lecture 6: Managing Directories

● Outline
  • Directories
  • Formatted I/O
  • Error Handling
  • The /proc Filesystem

● Reference
  • BLP: Chapters 3
Lecture 6: Managing Directories

Directory Scanning

- Use `opendir` to open a directory. Calls to `readdir` will return successive directory entries. `closedir` closes the directory.

```c
DIR *dp;
if ((dp = opendir(“.”)) == NULL)
    error_exit();
struct dirent *entry;
while((entry = readdir(dp)) != NULL) {
    cout << entry->d_name << endl;
}
if(closedir(dp) == -1)
    error_exit();
```
The `telldir` and `seekdir` routines allow you to reset a directory scan to a prior position.

See the program on pages 124-126 of your text for an example of using the directory routines to display a directory tree.

Note: Use `chdir()` and `getcwd()` to change and get current working directories.

Refer to `dirscan.cpp` for another example.
Lecture 6: Managing Directories

Formatted I/O

- The text discusses the C `stdio` routines. This library provides I/O routines similar to the UNIX routines: `fopen`, `fclose`, `fread`, and `fwrite`. In addition `fscanf` and `fprintf` provide formatted output. These routines are part of ANSI standard C.

- We will not cover these routines. You are encouraged to use C++ `iostreams` instead.
Lecture 6: Managing Directories

Formatted I/O

- Note that the **read** and **write** routines read and write binary data (**writenum1.cpp**). They DO NOT do **format** conversions (like **cin /cout** or **scanf / printf**).

- If you need formatted output, I recommend using a C++ **stringstream** or C **sprintf** to format binary data (see **writenum2.cpp** for a **stringstream** example).
Most system routines return -1 on error. The global `errno` variable is also set when an error occurs and indicates the type of error.

- **CHECK ALL ROUTINES FOR ERRORS!!!!**

- The `perror` routine decodes `errno` and displays a meaningful message on stderr.

- The `strerror` routine returns the `perror` message as a char string. It can be used to display custom error messages.
int main(int argc, char *argv[]) {
    if(argc != 2) {
        cerr << "usage: showerrs filename" << endl;
        exit(1);
    }
    int fd;
    if((fd = open(argv[1], O_RDWR)) == -1)
        { perror("showerrs"); exit(2); }
    if(close(fd) == -1)
        { perror("showerrs"); exit(3); }
    return 0;
}
Here is example output from running the program on the previous slide:

$ ./showerrs /etc/passwd
showerrs: Permission denied

$ ./showerrs /etc
showerrs: Is a directory

$ ./showerrs xxxxxxxxx
showerrs: No such file or directory
Linux uses several pseudo filesystems. These systems do not use up any space on the disk. They typically provide a means for the kernel to present information to applications or users.

We will briefly discuss only the /proc filesystem which provides an interface to kernel data structures. See man proc for information.
There is a /proc/[number] directory for each running process where number is the process ID. There are `cmdline`, `cwd`, `exe`, `environ`, etc. entries.

/proc/cpuinfo provides information about the CPU and architecture.

/proc/cmdline contains the arguments that are passed to the kernel at boot time.
Lecture 6: Managing Directories
The /proc Filesystem

- `/proc/net` contains several entries that provide information about various network layers.

- `/proc/filesystems` provides a list of all filesystems compiled into the kernel.

- `/proc/ide` and `/proc/scsi` provide information about corresponding devices.
The /proc Filesystem

- `/proc/sys` contains several entries corresponding to various kernel variables. These can not only be read, but also set to tune the kernel.

- `/proc/sys/fs/file-max` defines the system wide limit on the number of open files for all processes. This can be changed easily:

  ```
  echo 100000 > /proc/sys/fs/file-max
  ```