Lab 02 - The UNIX File System

Objectives

1. Explain what a file is in UNIX
2. List the 7 file types that are supported in UNIX
3. Learn about the file system organization on a UNIX system
4. Learn about some of the standard file system directories in UNIX
5. Write a C program that reads from a file

Pre-Reading

The following material contains commands and/or other information that could be essential to completing this and future labs. Students are strongly encouraged to take a look at the material below *before* coming to the lab, since knowing this material before you start will likely make task of completing the lab much faster and easier.

The UNIX File Concept

In UNIX, there are two important concepts you should know regarding **files**.

The first concept is:

Everything in UNIX is a file

Thus everything, including a network interface card, a disk drive, a keyboard, a printer, a simple/ordinary file, is treated as a file in UNIX. For example, the only way to access an I/O device in UNIX is to read or write to its corresponding **device file**. Similarly, the only way to communicate with a remote computer is to create a network connection file (called a **socket file**), and read/write to this file.

The second concept is:

A *file* is a sequence of bytes

Thus, everything in UNIX is seen as a simple stream of bytes. When you read from a device file, you're reading a sequence of bytes coming from that I/O device. When you send a message to a remote computer via an Internet connection, you're writing a sequence of bytes to that socket file.

The 7 File Types in UNIX

Simple/Ordinary File

Simple/ordinary files are used to store information and data on secondary storage devices, such as a disk. An ordinary file can contain a source program, an executable program, pictures, audio, video, and so on. UNIX treats all of these files the same, regardless of their content. That is, UNIX sees each of these files as a simple stream of bytes, nothing more. It's up to the application program to interpret the stream of bytes as something meaningful.

Directory

A directory file stores an array of directory entries, each of which corresponds to a file within that directory. Each directory entry has the form **<inode number, file name>**. The inode number is 4 bytes long, and is an index value for an array on the disk. An element of this array, known as an index node (or inode), contains file attributes such as file size, location on disk, owner, permissions, etc.

Symbolic Link

A symbolic link is a simple link to another file. It simply stores the absolute pathname of the file it points to.

Device Files (there are two types)

A device file (also called a special file) is a means of accessing hardware devices. Each hardware device is associated with at least one device file. A process accesses a device by reading or writing to its corresponding device file. There are two types of device files.
A **character device file** represents a character-oriented device, such as a keyboard.
A **block device file** represents a block-oriented device, such as a disk.

Named Pipe

A named pipe (also called a FIFO for First-In-First-Out) is one of several tools used for inter-process communication. That is, a named pipe is used to send sequences of bits between two running processes on the same computer. An application program sends data to another program by simply writing a stream of bytes to this file, and receives data by reading a stream of bytes from this file. The term FIFO simply means that sequences of bytes are received in the same order in which they were sent.

Socket

A socket can be used by processes on the same computer or on different computers to communicate with each other. The computers can be on a network or on the Internet.

Standard Files

When an application needs to perform an I/O operation on a file, it must first open the file and then issue the file operation (read, write, seek, etc.). UNIX automatically opens three files for every command it executes. The command reads input from one of these files and sends its output and error messages to the other two files. These files are called *standard files*: **standard input (stdin) file**, **standard output (stdout) file**, and **standard error (stderr) file**. By default, these files are attached to the terminal on which the command is executed. That is, the shell makes the command input come from the terminal keyboard and its output end error messages go to the console window.

The End-of-File (eof) Marker

Every UNIX file has an **end-of-file (eof) marker**. The commands that read their input from files read the eof marker when they reach the end of a file.

File System Organization

One of the most confusing differences between a UNIX-like operating system and Windows is the differences between the file systems. In both Windows and UNIX, a file system is organized hierarchically. That is, files are stored in directories, and these directories form a hierarchy. The top of a directory hierarchy is called the root directory.

The difference, however, is that a Windows system typically has multiple root directories, whereas a UNIX system has only one. In Windows, each root directory corresponds to a different physical drive. For example your main hard drive is usually **C:**, your optical drive is usually **D:**, and so on. In UNIX, the entire file system is organized into a single hierarchy. The top-most directory is called the **root** directory.

Some Standard UNIX Directories

Root Directory (/)

The top-most directory in a UNIX file system hierarchy is called the **root** (/) directory. All other pathnames in the system begin with (/), signifying that the given pathname starts at the root.

/bin

Also known as the binary directory, the /bin directory contains the binary (executable) files for *most* UNIX commands, including **`cat`**, **`ls`**, **`rm`**, and so on.

/dev

Also known as the device directory, the /bin directory contains the device files for the different I/O devices in the system.

/etc

This directory contains files for system administration (mostly system configuration files).

/users

This directory contains one subdirectory for every user of the system. The subdirectory for a given user is called the *home directory* of that user. For example, if your username is peter, then your home directory is /users/peter/

/usr

Also known as the UNIX System Resources directory, the /usr directory mostly contains binary (executable) files for commands, manual pages, and language libraries.

/proc

Also known as the process information pseudo filesystem, the /proc directory is a virtual directory that provide information from the kernel. By "virtual directory", we mean the directory doesn't actually exist on disk. Instead, it is created when the system starts, and the content of a given file within the directory generated *at the moment the file is read*, giving the *illusion* that the file is like any other file in the system.
For more information on the /proc directory, consult the man page **`man proc`**.
To see the contents of a given file in the /proc directory, use the **`cat`** command.

Reading a File in C

When programming in C, you can read and write to any file by sending and receiving a stream of bytes.

First, you must open the file using **`fopen`**. This returns a pointer to the file stream. By default, you automatically have access to pointers to the standard file streams **stdin**, **stdout**, and **stderr** via C variables with the *same name*.

Once you've opened the file, you can read a stream of bytes from it using **`fscanf`** (note that there are other standard C functions for reading and writing to files as well). For this function, you specify the file pointer, the type of data being read, and the variable in which to store the input. As always, lookup the **`man`** page for a standard C function that you don't understand. Note that, since **`fscanf`** allows you to specify the file pointer, entering **`stdin`** will allow you to read from the standard input stream.

When reading a file in C, you can check if the end-of-file marker has been reached using the standard C function **`feof`**.

When you're finished reading or writing to the file, you must close it using the **`fclose`** function.

The following is a simple program that reads from a file:

File: filereadex.c

```c
/* Contains function declarations for fopen, fscanf, fclose, and printf */
#include <stdio.h>

int main ()
{
      /* The unique identifier for the file to be read
         This is given to a Standard C Library function when specifying a file
         to read or write */
      FILE * fileID;

      /* A character array to store the string to be read from the file */
      char string[80];

      /* Opens the file for reading */
      fileID = fopen ("/proc/sys/kernel/foo","r");

      /* Reads in a string from a file */
      fscanf (fileID, "%s", string);

      /* Closes the file */
      fclose (fileID);

      /* Prints the string that was read from the file to the console */
      printf ("%s\n",string);

      return 0;
}
```

Command Line Arguments in C

In order to read command-line arguments in your C program, you must use the full declaration of the main function:

```c
int main (int argc, char *argv[])
```

The integer **argc** is the argument count. It is the number of arguments passed into the program from the command line, including the name of the program.

The array of character pointers is the listing of all the arguments. **argv[0]** is the name of the program, or an empty string if the name is not available. After that, every element number less than **argc** is a command line argument. You can use each **argv** element just like a string, or use **argv** as a two dimensional array. The last element in the array, **argv[argc]**, is a **null** pointer.

Try using the **printf** command to display elements in the **argv** array.

```c
printf ("%s\n", argv[0]);
```

Procedure

Write a C program that gathers the following information from the /proc directory:

- Processor Type.
- Kernel version.
- The amount of time since it was last booted.
- The amount of time the processor has spent in user mode, system mode, and idle time.
- The number of disk read/write requests made on the system.
- The number of context switches the kernel has performed.
- The time the system was last booted.
- The number of processes that have been created since the last boot.
- The amount of memory configured for this computer.
- The amount of memory currently available.

Your program (call it **`ksamp`**) will gather the above information every **x** seconds over a period of **y** seconds, where **x** and **y** are specified on the command line when you run your program. For example,

```
ksamp 2 60
```

specifies that your program should gather this information every 2 seconds, over a period of 60 seconds. Hence, the information would be gathered 30 times in total.

After all of the data samples have been gathered (i.e. after **y** seconds have elapsed), your program will output a well-formatted, easy-to-read report to the terminal, containing the ***average*** values for each item listed above.

Hints:

- There's no need to take averages of non-changing values, such as the Kernel version number.
- You may want to look up the man page for the **`sleep()`** system call.

Review Questions

There are no review questions for this lab.

Deliverables

1. All of the source and header files for your **`ksamp`** program.
2. A make file for building your program.