CS 452 Lab File System Interface: Operations

Overview

The purpose of this lab is to further investigate files and file systems. Specifically, the goal is to improve your understanding of typical file system API's (application programming interfaces) by utilizing the file locking and file linking system protocols.

Hand-in:

- A word document containing the requested information
- Any requested screenshots (uploaded as separate files)
- Program source code (no zip files)

File Locking

When investigating the shared memory problem, we performed synchronization of multiple processes via semaphores. The shared *file* problem is conceptually identical to the shared memory problem: how do we synchronize access to parts of a file to prevent overwriting or corruption of data when concurrent processes are accessing it? This problem lies at the heart of many applications - for example, database management systems and file access utilities.

As with the synchronization problem, the solution to sharing a file is conceptually similar to the shared memory solution: place a lock on all or part of the resource being accessed; in this case, a file. The lock prevents other users from modifying the file until it is unlocked. The familiar usage paradigm follows:

1) Obtain a lock on the file (or on a portion of the file)
2) Access/modify the file
3) Release the lock

The function for performing this activity is called **fcntl()** (file control). Take some time to study the man pages for this system call. The way the **fcntl()** function is used is reminiscent of the way semaphores operate using **semop()**: the fields of a structure are filled in with the appropriate values to specify a particular operation and then the function is called with the structure as one of its arguments. Based on the contents of the structure, the specified operation is executed. The structure to be used is of type **flock** (file lock).

The following program demonstrates the use of the **fcntl()** system call and the **flock** data structure:

sampleProgramOne

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>
#include <errno.h>

#define SIZE 30

int main(int argc, char *argv[])
{
  struct flock fileLock;
  int fd;
  char myBuffer[SIZE] = "// where did this come from?";

  if (argc < 2) {
    printf ("usage: filename\n");
    exit (1);
  }
  if ((fd = open (argv[1], O_RDWR)) < 0) {
    perror ("there is");
    exit (1);
  }

  fileLock.l_type = F_WRLCK;
  fileLock.l_whence = SEEK_SET;
  fileLock.l_start = 0;
  fileLock.l_len = 0;
  if (fcntl (fd, F_SETLK, &fileLock) < 0) {
    perror ("Unable to set file lock");
    exit (1);
  }

  write (fd, myBuffer, SIZE-2);

  sleep (10);

  close(fd);

  return 0;
}
```

Complete the following operations:

1. Study and compile sampleProgramOne

2. Using the source code file itself (sampleProgramOne) as the required command-line argument, execute the program
3. Carefully compare the modified source code file to the original sampleProgramOne, what exactly did sampleProgramOne do?
4. Re-execute the program, again using the source code as input. In a separate terminal window, immediately startup a second instance of your program (also using the same source code file as input). What happened, and why?

Modify sampleProgramOne so that it unlocks the file after sleeping. This modified Sample Program should now implement the normal resource usage paradigm:

a. Acquire a resource
b. Use the resource
c. Release the resource.

5. Create a second program (sampleProgramTwo) that *waits* until the lock is available and then reads and prints the first line of the source code file used as input
    - Note: read the man pages for the **fcntl()** system call to discover an easy way to wait
    - Test that your modified programs work together properly.
    - Submit your modified programs including screenshots of the execution.

Linking

The UNIX file system permits the creation of links, or "aliases" for files. There is a fundamental difference between hard links and symbolic (soft) links:

Hard: creates a directory entry using the new (aliased) pathname that has a pointer to an *existing* inode (for the *existing* file).
Symbolic: creates a directory entry with a new pathname and a pointer to a *new* inode, whose contents are the pathname of the file that will actually be referenced when the aliased filename is referenced.

Links are a useful way of maintaining different references to the same file. This mechanism can be used to provide users with their own access points to a shared file, or simply as a pathname abbreviation.

Read the man pages for the **ln** (link) command, which describes various ways of creating links. For file listing display purposes, specifically links, review the man pages describing the options available for the **ls** utility (this will help you distinguish links from regular files).

Perform the following operations:

- Create a new subdirectory. Make a copy of the original sampleProgramOne and name it ‘stuff’. Move the file stuff into your new subdirectory.

- Create both a hard link named hardStuff and a symbolic link named softStuff, both linking to the file ‘stuff’.
- Display (**cat**) the contents of the two link files (hardStuff and softStuff).
- Do a directory listing of the current directory, using 'long list' options (examine the listing carefully)

4. What are two ways to tell that a file is actually a symbolic link to another file?

Perform the following operations:

- Using '**ls**' with appropriate options, display the link count of the two link files

5. Why are the two link counts different?

- Display the sizes of the original file (stuff), and the two link files (hardStuff and softStuff)

6. What are the reported sizes of the 3 files? Why are the two link file sizes different?

Perform the following operations:

- Delete the file stuff in the subdirectory
- Display (**cat**) the contents of the two link files (hardStuff and softStuff)

7. What happened when you tried to display the link files? Explain.