CSc 332 (L) - Operating Systems

Lab – Spring 2018

File Management System Calls

We will work on this exercise in our lab of February 9,
and the completed work is due EOD February 16. Points
will be deducted for late submission.

- A system call is a call to the operating system's kernel requesting a service
- Provides an interface between a user program (or a process) and the operating system
- Can be viewed as special function calls
- A typical system call can be grouped into one of the following categories:
  - Process management
  - File management
  - Directory management
- Other system calls (Linux has over 300 different calls!), see `man syscalls`
- In this session, we will focus on system calls related to file management

`access` **system call**

- determines whether the calling process has access permission to a file
- checks for read, write, execute permissions and file existence
- takes two arguments
  - argument 1: file path
  - argument 2: `R_OK`, `W_OK` and `X_OK` corresponding to read, write, and execute permission
- return value is 0, if the process has all the specified permissions
- If the second argument is `F_OK`, the call simply checks for file's existence
- If the file exists but the calling process does not have the specified permissions, access returns -1 and sets `errno`

Example code snippet (incomplete) with system call (written in *C*)

Snippet 1: `check_file_permissions.c`

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>

int main (int argc' char* argv[])
{
    char* filepath argv[l];
    int returnval;

    // Check file existence
    returnval access (filepath' F-OK);
    if (returnval 0)
       printf ("\n %s exists\n"' filepath);
    else
    {
       if (errno ENOENT)
           printf ("%s does not exist\n"' filepath);
       else if (errno EACCES)
           printf ("%s is not accessible\n"' filepath);
       return 0;
    }


    // Check read access
    ...
    // Check write access
    ...
    return 0;
}
```

File Management System Calls

- To create new file, you use the `creat` system call. The named file is created as an empty file and opened for writing, and a positive integer, the open file identifier is returned. The file location is set to 0.
- In order to use a file, you first need to ask for it by name. This is called opening the file. The `open` system call creates an operating system object called an open file. The open file is logically connected to the file you named in the open system call. An open file has a file location (or file descriptor) associated with it and that is the offset in the file where the next read or write will start. The way in which the file is opened is specified by the *flags* argument: `O-RDONLY` to read; `O-wRONLY` to write; `O-RDwR` to both read and write; you can also do a bitwise OR with `O-CREAT` if you want the system to create the file if it doesn't exist already (e.g., `O-RDONLY|O-CREAT` creates and open in read mode). The system call returns the file descriptor of the new file (or a negative number if there is an error, placing the code into `errno`).
- After you open a file, you can use the `read` or `write` system calls to read or write the open file. Each read or write system call increments a file location for a number of characters read or written. Thus, the file is read (or/and written) *sequentially* by default. It returns the number of bytes read or written, or -1 if it ran into an error.

- The `lseek` system call is used to achieve random access into the file since it changes the file location that will be used for the next read or write.
- You close the open file using the `close` system call, when you are done using it. A return code 0 means the close succeeded.
- You delete the file from a directory using the `unlink` system call. A return code 0 means the unlink succeeded. If an error occurs in trying to delete a file, a negative integer is returned.

Table 1: System calls for file management

System Call	Parameters	Returns	Comment
open	name, flags	fd	Connect to open file
creat	name, mode	fd	Creates file and connect to open file
read	fid, buffer, count	count	Reads bytes from open file
write	fid, buffer, count	count	Writes bytes to open file
lseek	fid, offset, mode	offset	Moves to position of next read or write
close	fid	code	Closes or Disconnect open file
unlink	name	code	delete named file

An example to open an existing file

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <errno.h>

int main(int argc' char *argv[])
{
    int fd;
    if(2 !  argc)
    {
        printf("\n Usage    \n");
        return 1;
    }
    errno 0;
    fd open(argv[1]'O-RDONLY);

    if(-1 fd)
    {
        printf("\n open() failed with error [%s]\n"'strerror(errno));
        return 1;
    }
    else
    {
        printf("\n Open() Successful\n");
        /* open() succeeded' now one can do read operations on the file opened
```

```
        since we opened it in read-only mode. Also once done with processing'
        the file needs to be closed.*/
    }
    return 0;
}
```

The code above gives a basic usage of the open() function. Please note that this code does not show any read or close operations once the open() is successful. As mentioned in the comment in code, when done with the opened file, a close() should be called on the file descriptor opened. Now, lets run the code and see the output: First we try with a file (say, sample.txt) present in the same directory from where the executable was run. (Note, for files that are not in the same directory, an absolute path need to be specified).

```
$ ./open sample.txt

Open() Successful
```

How the system calls are processed?

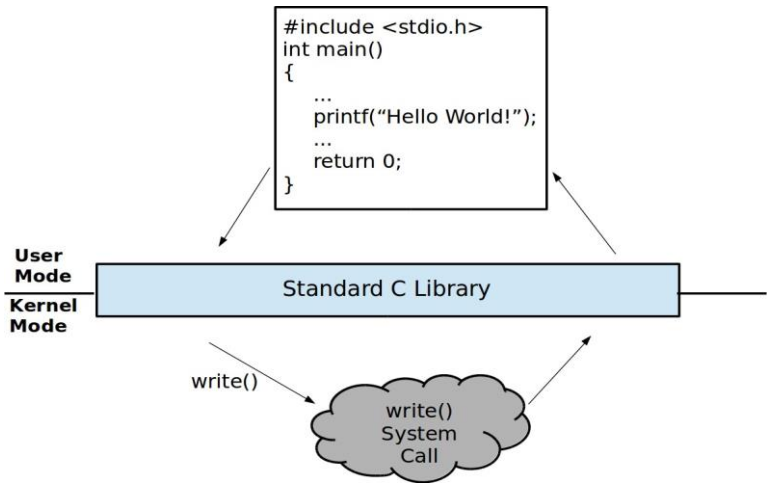

Figure 1: Standard C Library Example

Error codes

- System calls occasionally run into a problem
- Unix system calls handle this by returning some bad value (usually -1) and setting a global integer variable `errno` to some value that indicates the problem
- Subroutine `perror()` (it's not a system call - it's just a regular library function) that prints a meaningful error message to the standard error file

So here's how a typical segment might look.

```
fd open("filename"' O-RDONLY);
if(fd < 0)  /* ah' there's an error */
{
   printf("sorry' I couldn't open filename\n");
   perror("open"); /* This will explain why */
   return;
}
```

TASK 1

DUE: February 16, 2018 11:59 PM – 20 Points

LATE: February 23, 2018**, 11:59 PM – 12 Points**

0. (a) Extend code snippet 1 to check for read and write access permissions of a given file
   (b) Write a C program where `open` system call creates a new file (say, `destination.txt`) and then opens it. (Hint: use the bitwise OR flag)

1. UNIX `cat` command has three functions with regard to text files: displaying them, combining copies of them and creating new ones.

   Write a C program to implement a command called **displaycontent** that takes a (text) file name as argument and display its contents. Report an appropriate message if the file does not exist or can't be opened (i.e. the file doesn't have read permission). You are to use `open()`, `read()`, `write()` and `close()` system calls.

   NOTE: Name your executable file as *displaycontent* and execute your program as `./displaycontent file-name`

2. The `cp` command copies the source file specified by the *SourceFile* parameter to the destination file specified by the *DestinationFile* parameter.

   Write a C program that mimics the `cp` command using `open()` system call to open `source.txt` file in *read-only* mode and copy the contents of it to `destination.txt` using `read()` and `write()` system calls.

3. Repeat part 2 (by writing a new C program) as per the following procedure:

   (a) Read the next 100 characters from `source.txt`, and among characters read, replace each character '1' with character 'A' and all characters are then written in `destination.txt`

   (b) Write characters "XYZ" into file `destination.txt`

   (c) Repeat the previous steps until the end of file `source.txt`. The last read step may not have 100 characters.

General Instructions Use `perror` sub-routine to display meaningful error messages in case of system call failures. Properly close the files using `close` system call after you finish read/write. Learn to use `man` pages to know more about file management system calls (e.g, `man read`).

Submission Instructions

- You should use *only* file management system calls for file manipulation
- Use the given `source.txt`
- All the programs MUST be clearly indented and internally documented
- Make sure your programs compile and run without any errors
- Save all your programs with meaningful names and zip into a single folder as: task1_[your full name here].zip (e.g., `task1-peter barnett.zip`)
- Email your code with the subject line, "Task 1 - CSc 332 (L) – *full name*" (e.g., `Task1 - CSc 332 (L) - peter barnett`) to pbarnett1@ccny.cuny.edu

******