CMPSC 311 - Introduction to Systems Programming

Module: Input/Output

Professor Patrick McDaniel
Fall 2013
Assignment #3 (verification)

1. Download the latest start code (DONT OVERWRITE)
2. Make and run your program *without the verbose output* and log the output to a log (e.g., my.log)

   ./smsasim simple.dat -l my.log

3. Run the tool “verify”

   $ ./smsasim simple.dat -l my.log
   $ ./verify simple-output.log my.log
   Beginning diff check:

   Number correct / Total compared: 4097/4097
   Success.
   $

4. If the tool says, “success”, they the application is correct.
5. Note: see the correct output in simple-output.log, linear-output.log, and random-output.log
Input/Out

- Input/output is the process of moving bytes into and out of the process space.
  - terminal/keyboard (terminal IO)
  - devices /dev
  - kernel /proc
  - secondary storage (disk IO)
  - network (network IO)
Buffered vs. Unbuffered

• When the system is buffering
  ‣ It may read more than requested in the expectation you will read more later (*read buffering*)
  ‣ It may not commit all bytes to the target (*write buffering*)
Blocking vs. Nonblocking

- Non-blocking I/O
  - The call does not wait for the read or write to complete before returning (just does its best)
  - Thus a write/read may commit/return some, all, or none of the data requested
  - When fewer than request bytes are read/written this is called a short read or short write

- Note: how you program I/O operations is dependent on the blocking behavior of I/O you are using.
There are three default terminal channels.

- **STDIN**
- **STDOUT**
- **STDERR**

UNIX commands/programs for terminal output:

- **echo** - prints out formatted output to terminal STDOUT
  - e.g., `echo “hello world”`
- **cat** - prints out file (or STDIN) contents to STDOUT
  - e.g., `cat smsa_sim.c`
- **less** - provides a read-only viewer for input (or file)
  - e.g., `less smsa_sim.c`
IO Redirection

- Redirection uses file for inputs, outputs, or both
  - **Output redirection** sends the output of a program to a file (re-directs to a file), e.g.,
    - `echo "cmpsc311 output redirection" > this.dat`
      ```
      $ echo "cmpsc311 output redirection" > this.dat
      $ cat this.dat
      This is cmpsc311 output redirection
      ```
  - **Input redirection** uses the contents of a file as the program input (re-directs from a file), e.g.,
    - `cat < this.dat`
      ```
      $ cat < this.dat
      cmpsc311 output redirection
      ```
  - You can also do both at the same time, e.g.,
    - `cat < this.dat > other.dat`
Pipes

- Pipes take the output from one program and uses it as input for another, e.g.,
  - `cat this.dat | less`
- You can also chain pipes together, e.g.,
  - `cat numbers.txt | sort -n | cat`

```
3$ cat numbers.txt
14
21
7
4

$ cat numbers.txt | sort -n | cat
4
7
14
21
$
```
File IO

- File IO provides random access to a file within the filesystem:
  - With a specific “path” (location of the file)
  - At any point in time it has location pointer in the file
    - Next reads and writes will begin at that position
  - All file I/O works in the following way
    1. open the file
    2. read/write the contents
    3. close the file
Locating files for IO

- An **absolute path** fully specifies the directories and filename itself from the filesystem root “/”, e.g.,
  /home/mcdaniel/courses/cmpsc311-f13/this.dat
- An **relative path** is the directories and filename from (or relative to) the current directory, e.g.,
  ./courses/cmpsc311-f13/this.dat
courses/cmpsc311-f13/this.dat
  ./this.dat

- All of these references go to the same file!
FILE* based IO

- One of the basic ways to manage input and output is to use the FILE set of functions provided by clib.
  - The FILE structure is a set of data items that are created to manage input and output for the programmer.
  - An abstraction of “high level” reading and writing files that avoids some of the details of programming.
  - Almost always used for reading and writing ascii data

(gdb) p *file
$3 = {_flags = -72539008, _IO_read_ptr = 0x0, _IO_read_end = 0x0,
    _IO_read_base = 0x0, _IO_write_base = 0x0, _IO_write_ptr = 0x0,
    _IO_write_end = 0x0, _IO_buf_base = 0x0, _IO_buf_end = 0x0,
    _IO_save_base = 0x0, _IO_backup_base = 0x0, _IO_save_end = 0x0,
    _markers = 0x0, _chain = 0x7fffff7dd41a0 <_IO_2_1_stderr>, _fileno =
    7, _flags2 = 0, _old_offset = 0, _cur_column = 0,
    _vtable_offset = 0 '\000', _shortbuf = "", _lock = 0x6020f0, _offset
    = -1, __pad1 = 0x0, __pad2 = 0x602100, __pad3 = 0x0, __pad4 = 0x0,
    __pad5 = 0, _mode = 0, _unused2 = '\000' <repeats 19 times>}"
fopen()

- The fopen function opens a file for IO and returns a pointer to a FILE* structure:

  ```c
  FILE *fopen(const char *path, const char *mode);
  ```

- Where,
  - `path` is a string containing the absolute or relative path to the file to be opened.
  - `mode` is a string describing the ways the file will be used

  - For example,

    ```c
    FILE *file = fopen( filename, "r+" );
    ```

  - Returns a pointer to FILE* if successful, NULL otherwise

- You don’t have to allocate or deallocate the FILE* structure
The fopen function opens a file for IO and returns a pointer to a FILE* structure:

```c
FILE *fopen(const char *path, const char *mode);
```

Where,

- `path` is a string containing the absolute or relative path to the file to be opened.
- `mode` is a string describing the ways the file will be used.

For example,

```c
FILE *file = fopen( filename, "r+" );
```

A FILE* structure is also referred to as a **stream**.

Returns a pointer to FILE* if successful, NULL otherwise.

You don’t have to allocate or deallocate the FILE* structure.
fopen modes

• "r" - Open text file for reading. The stream is positioned at the beginning of the file.
• "r+" - Open for reading and writing. The stream is positioned at the beginning of the file.
• "w" - Truncate file to zero length or create text file for writing. The stream is positioned at the beginning of the file.
• "w+" - Open for reading and writing. The file is created if it does not exist, otherwise it is truncated.
• "a" - Open for appending (writing at end of file). The file is created if it does not exist.
• "a+" - Open for reading and appending (writing at end of file). The file is created if it does not exist.
There are two dominant ways to read the file, `fscanf` and `fgets`

- `fscanf` reads the data from the file just like `scanf`, just reading and writing, e.g.,

```c
if ( fscanf(file, "%d %d %d\n", &x, &y, &z ) == 3 ) {
    printf("Read coordinates [%d,%d,%d]\n", x, y, z );
}
```

- `fgets` reads a line of text from the file, e.g.,

```c
if ( fgets(str,128,file) != NULL ) {
    printf("Read line [%s]\n", str );
}
```
Reading the file

- There are two dominant ways to write the file, **fprintf** and **fgets**
  - **fprintf** writes the data to the file just like printf, just reading and writing, e.g.,
    
    ```c
    fprintf( file, "%d %d %d\n", x, y, z );
    ```

  - **fgets** writes the a line of text to the file, e.g.,
    
    ```c
    if ( fgets(str,file) != NULL ) {
        printf( "wrote line [%s]\n", str );
    }
    ```
• FILE*-based IO is buffered
• `fflush` attempts to reset/the flush state

\[
\text{int } \text{fflush}(\text{FILE } * \text{stream});
\]

‣ FILE*-based writes are buffered, so there may be data written, but not yet pushed to the OS/disk.
• `fflush()` forces a write of all buffered data

‣ FILE*-based reads are buffered, so the current data (in the process space) may not be current
• `fflush()` discards buffered data from the underlying file

• If the stream argument is NULL, `fflush()` flushes all open output streams
fclose() closes the file and releases the memory associated with the FILE* structure.

```c
fclose( file );
file = NULL;
```

**Note:** `fclose` implicitly flushes the data to storage.
int show_fopen( void ) {

    // Setup variables
    int x, y, z;
    FILE *file;
    char *filename = "/tmp/fopen.dat", str[128];
    file = fopen( filename, "r+" );

    // open for reading and writing
    if ( file == NULL ) {
        fprintf( stderr, "fopen() failed, error=%s\n", strerror(errno) );
        return( -1 );
    }

    // Read until you reach the end
    while ( !feof(file) ) {
        if ( fscanf( file, "%d %d %d\n", &x, &y, &z ) == 3 ) {
            printf( "Read coordinates [%d,%d,%d]\n", x, y, z );
        }

        if ( fgets(str, 128, file) != NULL ) {
            printf( "Read line [%s]\n", str );
        }
    }
}
// Now add some new coordinates
x = 21;
y = 34;
z = 98;
fprintf( file, "%d %d %d\n", x, y, z );
printf( "Wrote %d %d %d\n", x, y, z );
if ( fputs(str,file) >= 0 ) {
    printf( "wrote line [%s]\n", str );
}
fflush( file );

// Close the file and return
fclose( file );
return( 0 );

$ cat /tmp/fopen.dat
1 2 3
4 5 6
11 12 14
16 17 23
$ ./io
This is cmpsc311, IO example
Read coordinates [1,2,3]
Read line [11 12 14]
Read coordinates [16,17,23]
Wrote 21 34 98
wrote line [11 12 14]

$ cat /tmp/fopen.dat
1 2 3
4 5 6
11 12 14
16 17 23
21 34 98
11 12 14
$
open() 

• The `open` function opens a file for IO and returns an integer `file handle`:

```c
int open(const char *path, int flags, mode_t mode);
```

• Where,
  ‣ `path` is a string containing the absolute or relative path to the file to be opened.
  ‣ `flags` indicates the kind of open you are requesting
  ‣ `mode` sets a security policy for the file

• `open()` returns a `file handle`
open() flags

- The “mode” to open with
  - O_RDONLY - read only
  - O_WRONLY - write only
  - O_RDWR - read and write

- Others
  - O_CREAT - If the file does not exist it will be created.
  - O_EXCL Ensure that this call creates the file, an fail otherwise (fail if already exists)
  - O_TRUNC - If the file already exists it will be truncated to length 0.

Note: You bitwise or (|) the options you want
Access Control in UNIX

• The UNIX filesystem implements *discretionary access control* through file permissions set by user
  ‣ The permissions are set at the discretion of the user
• Every file in the file system has a set of bits which determine who has access to the files
  ‣ *User* - the owner is typically the creator of the file, and the entity in control of the access control policy
  ‣ *Group* - a set of users on the system setup by the admin
  ‣ *World* - the set of everyone on the system

• **Note**: this can be overridden by the “root” user
UNIX filesystem rights …

- There are three rights in the UNIX filesystem
  - **READ** - allows the subject (process) to read the contents of the file.
  - **WRITE** - allows the subject (process) to alter the contents of the file.
  - **EXECUTE** - allows the subject (process) to execute the contents of the file (e.g., shell program, executable, …)

- **Q**: why is execute a right?
- **Q**: does read implicitly give you the right to execute?
• Really, this is a bit string encoding an access policy:

rwx  rwx  rwx

World

Group

Owner

• And a policy is encoded as “r”, “w”, “x” if enabled, and “-” if not, e.g,

rwxrw---x

• Says user can read, write and execute, group can read and write, and world can execute only.
UNIX Access Policy

- Really, this is a bit string encoding an access policy:

\[
\text{rwx rwx rwx}
\]

- And a policy is encoded as “r”, “w”, “x” if enabled, and “-” if not.

Says user can read, write and execute, group can read and write, and world can execute only.

```bash
$ ls -l .
total 52
-rw-rw-r-- 1 professor mcdaniel 12 Oct 10 14:18 fopen.dat
-rwxrwxr-x 1 professor mcdaniel 12058 Oct 10 15:42 io
-rw-rw-r-- 1 professor mcdaniel 1176 Oct 10 15:42 io.c
-rw-rw-r-- 1 professor mcdaniel 88 Oct 10 14:17 Makefile
-rw-rw-r-- 1 professor mcdaniel 15633 Oct 10 10:46 mmap.dat
-rw-rw-r-- 1 professor mcdaniel 50 Oct 10 10:58 other.dat
-rwxrwxr-x 1 professor mcdaniel 154 Oct 10 10:58 redirect.sh
-rw-rw-r-- 1 professor mcdaniel 50 Oct 10 10:58 this.dat
$```
Setting an access policy

• Specify a file access policy by bit-wise ORing (|):
  - S_IRWXU 00700 user (file owner) has read, write and execute
  - S_IRUSR 00400 user has read permission
  - S_IWUSR 00200 user has write permission
  - S_IXUSR 00100 user has execute permission
  - S_IRWXG 00070 group has read, write and execute permission
  - S_IRGRP 00040 group has read permission
  - S_IWGRP 00020 group has write permission
  - S_IXGRP 00010 group has execute permission
  - S_IRWXO 00007 world has read, write and execute permission
  - S_IROTH 00004 world has read permission
  - S_IWOTH 00002 world has write permission
  - S_IXOTH 00001 world has execute permission
Putting it together ...

• So an open looks something like ...

```c
// Setup the file for creating and open
flags = O_WRONLY|O_CREAT|O_EXCL;  // Create a NEW file (no overwrite)
mode = S_IRUSR|S_IWUSR|S_IRGRP;  // User can read/write, group read
fhandle = open( filename, flags, mode );
if ( fhandle == -1 ) {
    fprintf( stderr, "open() failed, error=%s\n", strerror(errno) );
    return( -1 );
}
```

Q: But how is an `int` returned by `open()` a file?
A **file descriptor** is an index assigned by the kernel into a table of file information maintained in the OS:

- The file descriptor table is unique to each process and contains the details of open files.
- File descriptors are used to reference when calling the I/O system calls.
- The kernel accesses the file for the process and returns the results in system call response.
Reading and Writing

• Primitive reading and writing mechanisms that only process only blocks of opaque data:

  ssize_t write(int fd, const void *buf, size_t count);
  ssize_t read(int fd, void *buf, size_t count);

• Where \texttt{fd} is the file descriptor, \texttt{buf} is an array of bytes to write from or read into, and \texttt{count} is the number of bytes to read or write.

• In both \texttt{read()} and \texttt{write()}, the value returned is the number of bytes read and written.
  ‣ Be sure to always check the result

• On reads, you are responsible for supplying a buffer that is large enough to put the output into.
close()

- close() closes the file and deletes the file’s entry in the file descriptor table

```c
close( fhandle );
fhandle = -1;
```

**Note:** Always reset your file handles to -1 to avoid use after close.
```c
int show_open( void ) {

    // Setup variables
    char *filename = "/tmp/open.dat";
    int vals[1000] = { [0 ... 999] = 0xff }, vals2[1000];
    int fhandle, flags;
    mode_t mode;

    // Setup the file for creating and open
    flags = O_WRONLY|O_CREAT|O_EXCL; // Create a NEW file (no overwrite)
    mode = S_IRUSR|S_IWUSR|S_IRGRP; // User can read/write, group read
    fhandle = open( filename, flags, mode );
    if ( fhandle == -1 ) {
        fprintf( stderr, "open() failed, error=%s\n", strerror(errno) );
        return( -1 );
    }

    // Now write the array to the file
    if ( write(fhandle, (char *)vals, sizeof(vals)) != sizeof(vals) ) {
        fprintf( stderr, "write() failed, error=%s\n", strerror(errno) );
        return( -1 );
    }

    close( fhandle );
    fhandle = -1;
}
```
// Setup the file for reading
flags = O_RDONLY;  // Read an existing file
fhandle = open( filename, flags, 0 );
if ( fhandle == -1 ) {
    fprintf( stderr, "open() failed, error=%s\n", strerror(errno) );
    return( -1 );
}

// Now read the array from the file
if ( read(fhandle, (char *)vals2, sizeof(vals2)) != sizeof(vals2) ) {
    fprintf( stderr, "read() failed, error=%s\n", strerror(errno) );
    return( -1 );
}
close( fhandle );
return( 0 );

$ ./io
$ $ od -x -N 256 /tmp/open.dat
00000000 00ff 0000 00ff 0000 00ff 0000 00ff 0000
* 
0000400
fopen() vs. open()

• Key differences between fopen and open
  ‣ fopen provides you with buffering IO that may or may not turn out to be a faster than what you're doing with open.
  ‣ fopen does line ending translation if the file is not opened in binary mode, which can be very helpful if your program is ever ported to a non-Unix environment.
  ‣ A FILE * gives you the ability to use fscanf and other stdio functions that parse out data and support formatted output.

• IMO: use FILE* style I/O for ASCII processing, and file handle I/O for binary data processing.
A parting note ... 

• Each of the styles of I/O requires a different set of include files
  ▪ FILE* requires:
  ▪ file handle I/O requires:
A parting note ...

• Each of the styles of I/O requires a different set of include files
  ‣ FILE* requires:
    
    ```c
    #include <stdio.h>
    ```
  ‣ file handle I/O requires:
• Each of the styles of I/O requires a different set of include files
  ‣ FILE* requires:
    ```
    #include <stdio.h>
    ```
  ‣ file handle I/O requires:
    ```
    #include <sys/types.h>
    #include <sys/stat.h>
    #include <fcntl.h>
    #include <unistd.h>
    ```