CpSc 360: Distributed and Network Programming

Unix I/O Programming

James Wang

File I/O

http://developers.sun.com/solaris/articles/fileio.html

2

File Descriptors

- File descriptors are the fundamental way of accessing files.
- A file descriptor, which is a small positive integer, is actually the offset into the process' process file table.
- Each process has a process file table associated with it, and it is this table that provides a mapping between the process' idea of the file, and the kernel's.
- Each of the basic file I/O functions takes a file descriptor as an argument, or returns one.

3

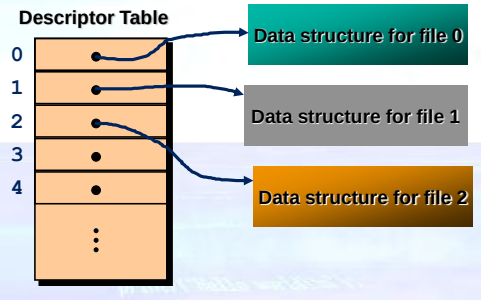
Standard FDs

- There are three file descriptors that the shell opens: standard input, standard output, and standard error.
- These are conventionally assigned to descriptors 0, 1, and 2 respectively.
- Instead of these magic numbers, newer programs should use the POSIX.1 constants

  STDIN_FILENO,
  STDOUT_FILENO,
  STDERR_FILENO,
  which are defined in <unistd.h>.

4

Unix Descriptor Table



5

The open Function

- To open a file, use the open function.

  int open (const char *path, int oflag, /* mode_t mode */...);
  - The file you want to open is pointed to by path.
  - the mode you want to open it for, as well as other flags, is specified in *oflag*.
  - If the call to open causes a file to be created, it is created with the permissions in *mode*, modified by the process's umask.
- If the call to open is successful, the file's file descriptor is returned; otherwise, -1 is returned, and errno is set appropriately.
- The file descriptor that is returned is guaranteed to be the lowest unused one available.

6

Option Flags

- **Open a file for reading, writing, or both, by setting *oflag* to one of the following mutually exclusive constants.**
  - O_RDONLY: Open the file for reading only.
  - O_WRONLY: Open the file for writing only.
  - O_RDWR: Open the file for reading and writing.
- **More option flags:**
  - O_APPEND: Sets the file offset to the end of the file prior to each write. This is useful if, for example, more than one process is updating the file, as the writes won't overwrite each other.
  - O_CREAT: If the file exists, this flag has no effect, unless O_EXCL is also specified. Otherwise the file is created, with the file's owner set to the effective user ID of the process. The access permission bits of the file are set by *mode*, as modified by the process' umask.
  - O_TRUNC: If the file exists, and is successfully opened for writing, setting this flag will cause the file's length to be truncated to 0 bytes. Any data that the file contains is discarded.

7

The creat Function

- **The creat system call is another way of creating a file.**

  int creat(const char *path, mode_t mode);

  - The name of the file to create is pointed to by *path*.
  - If the file is successfully created, its permission bits are set to *mode*, as modified by the process' umask;
  - if the file can't be created, creat returns -1 , and errno is set.
  - The creat system call is equivalent to

    open(path, O_WRONLY | O_CREAT | O_TRUNC, mode);

8

The close function

- **The close function closes an open file.**

  int close (int fd);

- ***fd* should be a file descriptor that was previously returned by open, creat, dup, (or dup2), or pipe. When a process closes a file, any locks that it may have on the file are released.**
- **When a process exits, all of its files are automatically closed, a fact taken advantage of by many programmers. It is considered good programming practice to close a file when you are finished with it, as file descriptors are a finite resource.**

9

The lseek and llseek Functions

- **Every open file has an associated *file offset*, which determines where the next read or write operation will start.**
- **The file offset is set to 0 when a file has been opened, and is automatically increased after each successful read or write.**
- **Reads from a file descriptor start from the current file offset, as do writes, unless the O_APPEND flag was set when the file was opened (in which case the file offset gets set to the end of the file at the start of every write).**
- **You can change the file offset for an open file by using either the lseek or llseek functions.**

  off_t lseek (int fildes, off_t offset, int whence);

  offset_t llseek (int fildes, offset_t offset, int whence);

10

Offset in lseek and llseek

- **How the value of *offset* is interpreted depends on the value of the *whence* argument.**
  - If *whence* is SEEK_SET, the file pointer is set to *offset* bytes.
  - If *whence* is set to SEEK_CUR, the file pointer is set to its current location plus *offset*.
  - If *whence* is SEEK_END, the file pointer is set to the end of the file plus *offset*.
- **The constants SEEK_SET, SEEK_CUR, and SEEK_END are defined in <unistd.h>, and have the values of 0, 1, and 2 respectively, for compatibility with older code.**

11

The tell Function

- **The tell function is used to get the current file offset for a file descriptor.**

  off_t tell (int fd);

- **Notice that the return type is an off_t (which is a 32 bit quantity), rather than an offset_t (which is a 64 bit quantity).**
  - This means that you cannot safely use it for files that were opened with the O_LARGEFILE flag specified, because the file's offset may be too large to fit into an off_t.

12

The read and pread Functions

- **Use the read and pread functions to read data from an open file.**

```
ssize_t read (int fildes, void *buf, size_t nbyte);
ssize_t pread (int fildes, void *buf, size_t nbyte, off_t offset);
```

- **The read function reads up to *nbyte* bytes from the open file referred to by *fd* into the buffer pointed to by *buf*. If the read is successful, the number of bytes read is returned, unless you are at the end of file, in which case 0 is returned.**
- **The pread function is identical to read, except that pread read operations start at the specified *offset*, without changing the file pointer.**
  - Attempting to perform a pread on a file that is incapable of seeking will result in an error.

13

The write and pwrite Functions

- **Use the write and pwrite functions to write data to an open file.**

```
ssize_t write (int fildes, void *buf, size_t nbyte);
ssize_t pwrite (int fildes, void *buf, size_t nbyte, off_t offset);
```

- **The write function writes up to *nbyte* bytes to the open file referred to by *fd* from the buffer pointed to by *buf*. If the write is successful, the number of bytes written is returned.**
- **The pwrite function is identical to write, except pwrite write operations start at the given *offset*, without changing the file pointer.**
  - As with pread, attempting to perform a pwrite on a file that is incapable of seeking results in an error.

14

Error Handling

15

System Calls and Errors

- **In general, systems calls return a negative number to indicate an error.**
  - We often want to find out what error.
  - Servers generally add this information to a log.
  - Clients generally provide some information to the user.
- **Whenever an error occurs, system calls set the value of the global variable `errno`.**
  - You can check errno for specific errors.
  - You can use support functions to print out or log an ASCII text error message.

16

How to use errno?

- `errno` **is valid only after a system call has returned an error.**
  - System calls don't *clear* **`errno`** on success.
  - If you make another system call you may lose the previous value of **`errno`**.
    - **`printf`** makes a call to **`write`**!
- **Support routines:**

`void perror(const char *string);`

The perror() function produces a message on the standard error output (file descriptor 2) describing the last error encountered during a call to a system or library function.

`char *strerror(int errnum);`

The strerror() function maps the error number in errnum to an error message string, and returns a pointer to that string.

17

Error codes

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

Actually defined in "/usr/include/sys/errno.h".

Sample error codes:

```
......
/* BSD Networking Software */
    /* argument errors */
#define ENOTSOCK       95    /* Socket operation on non-socket */
#define EDESTADDRREQ   96    /* Destination address required */
#define EMSGSIZE       97    /* Message too long */
#define EPROTOTYPE     98    /* Protocol wrong type for socket */
#define ENOPROTOOPT    99    /* Protocol not available */
#define EPROTONOSUPPORT 120  /* Protocol not supported */
#define ESOCKTNOSUPPORT 121  /* Socket type not supported */
#define EOPNOTSUPP     122   /* Operation not supported on socket */
#define EPFNOSUPPORT   123   /* Protocol family not supported */
#define EAFNOSUPPORT   124   /* Address family not supported by */
......
```

18

General Strategies

- Include code to check for errors after every system call.
- Develop "wrapper functions" that do the checking for you.
- Develop layers of functions, each hides some of the error-handling details.
- Example wrapper:

```
int Open( const char* path, int oflag, mode_t mode) {
        int n;
        if ( (n=open(path, oflag, mode)) < 0 )) {
                perror("Fatal error on file open");
                exit(1);
        }
        return(n);
}
```

19

How to handle errors?

- How do you know what should be a fatal error (program exits)?
  - Common sense.
  - If the program can continue – it should.
  - Example – if a server can't create a socket, or can't bind to it's port - there is no sense continuing…
- Retry if it is necessary.

20

Use of Wrappers

- Use wrappers as the authors did in the textbook can make code much more readable.
- There are always situations in which you cannot use the wrappers
  - Sometimes system calls are "interrupted" (EINTR) – this is not always a fatal error !
- If you use the code from the book for your projects, you must understand it!
- The library of code used in the text is extensive:
  - Wrappers call custom error handing code.
  - Custom error handling code make assumptions about having other custom library functions.
  - …

21

Layering Function

- Instead of simple wrapper functions, you might develop a *layered system*.
- Developing general functions that might be re-used in other programs is obviously "a good thing".
- Layering is beneficial even if the code is not intended to be re-used:
  - hide error-handling from "high-level" code.
  - hide other details.
  - often makes debugging easier.

22

Summary of Error Handling

- There is no *best approach* that will fit everybody.
- Do what works for you. But doing nothing won't work for anybody.
- Make sure you check *all* system calls for errors!!!!
  - Not checking can cause your program malfunction in certain system conditions.
  - Not checking can lead to security problems!
  - Not checking can lead to bad grades on homework projects!

23

Program Debugging

24

Who Need Debug?

- **Nobody writes a perfect program at the first round.**
- **You need assert yourself.**

```
#include <assert.h>

void assert(int expression);
```

- **The assert() macro inserts diagnostics into applications. When executed, if expression is FALSE (zero), assert() aborts after printing the following error message:**

  Assertion failed: expression, file xyz, line nnn

25

Stubs and Drivers

- **It is good practice to test each function you write with a separate short "driver" program before using it in other longer or more complex programs.**
- **The driver program might typically be a simple "while" loop that allows you to pass various parameters to the function from the keyboard, until you are satisfied that the function works properly.**
- **If you are unsure about the overall control flow of your main program, it is often convenient to first test it with "stub" functions, which merely return an arbitrary value of the correct data type.**
  - This technique is especially useful when using a top down design method, and facilitates procedural or functional abstraction.

26

Some Common Errors

- **Typing "=" instead of "==" (and vice-versa).**
  - When comparing a variable with a constant, it is possible to avoid some of these errors by getting into the habit of putting the constant on the left.
- **Omitting a ";" particularly before a "}".**
- **Forgetting the "()"s when calling a function with no parameters.**
- **Putting the wrong case in identifiers.**
- **Omitting brackets round a conditional test.**
- **Using integer division when floating is required (and vice versa).**

27

More Common Errors

- **Getting the test and increment the wrong way round in a "for" statement.**

  E.g. don't write for(int i=0; i++; i<10)
- **Using "<" in a for loop when "<=" is required (and vice versa).**
- **Assuming an array "a" of length L has an element a[L].**
- **Having a "#include" directive in a header file and source file. These can be in one place or the other, but not both.**
- **Doing a "#include" on another source file rather than a header.**

28

Debugging Very Bad Programs

- **If your program is very bad, the best way to debug it is to:**
  - throw it away.
  - ask yourself what you did in your programming courses.
  - don't panic.
  - read C programming book or online tutorial.
  - practise! practise! practise!!!!!!

Call your instructor for help!!!!

29

GNU debugger gdb

- **The gdb debugger is a powerful tool for tracking down errors in your programs.**
- **You can run gdb from the unix prompt in CS machines by typing gdb**
- **To include debugging information remember to compile your program with the -g option.**

30

gdb commands

r	run program until breakpoint, natural termination or fatal error
c	continue/resume program execution
s	step until next line (will step into function calls)
n	continue until next line (steps over function calls) finish continue until function in current stack frame returns break set breakpoint (argument can be function or line number)
watch	set watchpoint (argument is an expression; debugger will stop execution whenever expression changes)
info break	list all breakpoints and watchpoints
where	show stack trace
up	move to higher stack frame
down	move to lower stack frame
print	displays the values of a variable in the current stack frame
quit	quits the debugger

Terminal I/O Issues

How do you send EOF?

- **The answer depends on what kind of computer you're using.**
  - On Unix and Unix-related systems, it's almost always control-D.
  - On MS-DOS machines, it's control-Z followed by the RETURN key.
  - Under Think C on the Macintosh, it's control-D, just like Unix.
  - On other systems, you may have to do some research to learn how to send EOF.

What EOF and CTRL-D Really Mean

- **EOF is not a character.**
  - **EOF is an integer error code returned by some functions to indicate that they've reached the end of input.**
  - **So, don't try to treat "EOF" as a character in your programs, because it's not. Also, don't just assume that the output of getchar() or what-have-you will be a character, because it might be an error code outside of the character range. If it says it returns int, store its result in an int.**
- **CTRL-D is not actually sent to your program.**
  - **When you run your program, your program is given a file handle called "stdin" or a file descriptor called "STDIN_FILENO" . It neither knows nor cares what's on the other end of it. The command shell that you run the program from sets up this stream and passes input to the program, based on what you type.**

    CTRL-D is character that you send to the shell to tell it that you want to stop sending information to the program. The shell sees the CTRL-D, and immediately cuts off input to the program. Subsequent reads by the program get an "End Of File" error, because the stream is closed. The CTRL-D character is never sent to the program (the shell interprets it as a command and acts on it).

Keyboard Input vs. File Input

- **The program can't tell the difference between keyboard input and file input.**
  - **As described in the previous topic, the program just gets a file handle that it can read data from. It gets a stream of characters until whatever's on the other end cuts off the stream.**
- **When you're using keyboard input, the shell makes it pretty hard to signal "end of input".**
  - **Pressing "enter" to get to a new line, then CTRL-D, will do it. I'm told that pressing CTRL-D twice in the middle of a line will do it also. All of these are quirks of the shell - they're not the fault of your program. However, they'll still make it more difficult for you to test some input cases with your programs. Remember that you can always use "cat" to send a text file to your program as input.**

Newlines

- **What is a Newline?**
  - A newline is a character (like any other character), that's interpreted as a positioning command by most programs that display text. When you hit "enter" when testing a program, all that happens is that your program receives character 0x0a (line feed) in its input stream.

    Windows is an exception. See below.
- **Newlines under Windows**
  - Windows uses "CRLF" in text files to denote end-of-input, instead of just "LF". This will cause problems if you transport files between Windows and Unix. Under Windows, you may see everything mashed on to one line with funny characters where the line breaks should be. Under Unix, you may see "^M" everywhere in your input (the carriage-return characters). Some programs are smart enough to convert when reading files like this. Others aren't.

    Be aware of this when transporting source files between these systems, and don't assume that line breaks will always be one character long if you're going to be reading files produced on a Windows machine.

Input Buffer

- **Input from the keyboard is line-buffered.**
  - **The shell doesn't send what you type to your program immediately.**
  - **Instead, it buffers it internally until it sees a newline or until its internal buffer gets full.**
  - **If you're wondering why your program seems to wait until you hit enter even when it should act immediately, this is probably the reason.**

How to Get Input Characters Immediately?

- **Finally, don't be disappointed (as I was) when you expect your program to respond your every key strokes.**
- **Instead, the computer responds only when you hit RETURN, a full line of characters is made available to your program.**
- **You may wonder how a program could instead read a character right away, without waiting for the user to hit RETURN.**
- **How to do it depends on which operating system you're using:**
  - **In Windows:** use getch(). (but need to include <conio.h> or <curses.h>)
  - **In Unix:** I have no idea at this moment because how to read a character right away is one of the things that's not defined by the C language, and it's not defined by any of the standard library functions, either. However, you can clearly use some customer made libraries to achieve this.

http://www.pwilson.net/kbhit.html