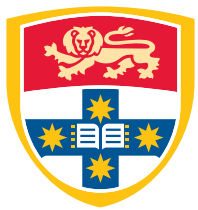
Introduction to Programming Advanced 2020 S2

Dr. John Stavrakakis

School of Computer Science, University of Sydney



Copyright Warning

Lecture 6:

IO

“What are files?”

Disk storage peripherals provide persistent storage with a low-level interface

- Fixed size blocks
- Numeric Addresses

Operating system arranges this into an *abstraction as files*

- Files can be of variable length
- Have names, and metadata (owner, permissions etc)
- Have an external structure (directories/folders)

Files

There are three operations that can be performed on a file:

- Read - get data from a file
- Write - put data into a file
- Execute - spawn a process using the contents of the file as the instructions

These operations are handled by System Calls

Files

Devices are often represented as files

Software can then read or write to the file to access the device

For example, a keyboard input might just be the output of the keyboard stored in a file

If a file can be a physical device, then it is not fixed in size or in its behaviour

Low Level File Descriptors

Low level I/O is performed on file descriptors

These are integers that reference members of the file table, which itself references an inode on disc.

Inodes are file system objects and may be either files, or directories.

Opening a file creates a new entry in the file table referencing an inode, and assigns the next consecutive file descriptor to that entry.

Syscalls

When a process is started, file descriptor 0 is standard input, 1 is standard output, 2 is standard error.

Our system calls operate on these file descriptors.

- `creat, open, close` - Creates inodes, opens and closes file descriptors
- `read, write` - Reads and writes to or from a file descriptor
- `ioctl` - Performs IO requests on an open file descriptor
- `umask` - Modifies permissions on a file

It is important to remember that read and write act on fixed sizes of memory. For example, reading 100 bytes from standard input to a buffer would look like:

```
ssize_t n_bytes = read(0, buffer, 100);
```

Flags

We previously mentioned a file table, this table handles our access to the inode containing the data for the file.

When we open a file, we need to tell the file table what access is required.

This is done using the flags argument of the `open` syscall.

- `O_RDONLY` - Read only mode
- `O_WRONLY` - Write only mode
- `O_RDWR` - Read Write
- `O_CREAT` - Creates the file if it doesn't exist

Flags

The flags themselves are integers and are represented using some number of 1 bits in the integer.

We can compose multiple flags in to a single argument using the bitwise OR operation to create a single flag argument with the correct bits set. Here `fd` is our file descriptor.

```
int fd = open("file.txt", O_RDONLY | O_CREAT);
```

It's worth mentioning that all files that you open should then be closed:

```
close(file_descriptor);
```

read

Given our file descriptor, we can read `n_bytes` from it to a buffer using the `read` syscall.

```
ssize_t read(int fd, void* buf, size_t n_bytes)
```

The integer returned is the number of bytes read from the file descriptor, if the value is 0 then the end of the file has been reached as there is no longer any data to read.

This syscall is very primitive; when reading characters to a buffer as a string you will need to include your own null terminator.

write

Similarly we have our write function

```
ssize_t write(int fd, const void* buf, size_t n_bytes)
```

Here we write `n_bytes` from `buf` to our file descriptor `fd`.

When working with strings, it's a good idea to use `strlen` to ensure that the correct number of bytes are being written.

seek

When accessing a file we do so using a file offset that tracks a position within the file.

When reading from or writing to a file, the position of the file offset is updated by the required number of bytes.

We can also modify the file offset using the `lseek` sys call.

seek

Our `lseek` call takes the form

```
off_t lseek(int fd, off_t offset, int whence)
```

Here `fd` is the file descriptor, `offset` is the number of bytes and `whence` specifies one of three modes:

- `SEEK_SET` - Sets from the start of the file plus `offset` bytes
- `SEEK_CUR` - Sets from the current file offset position plus `offest` bytes
- `SEEK_END` - Sets from the end of the file plus `offset` bytes

`lseek` returns the number of bytes it has offset the file by.

Using these flags we can re-read relevant sections of the file rather than having to re-open the file.

Files Using the C Library

While we could use the syscalls to continue to access our files, C provides some of its own functions.

A *stream* is associated with a file descriptor

- May support a file position indicator
- May be binary or not (ASCII, multibyte etc)
- Can be open, closed, flushed!
- Can be unbuffered, fully buffered or line buffered

`stdio.h` contains may standard IO functions and definitions for accessing files and streams.

feof

Unlike our file offset, which can merrily pass the end of the file, our stream has an end of file indicator to flag when the end of the file has been reached.

We can check whether we've reached the end of the file with the `feof` function, which tests for the end of file indicator.

```
int feof(FILE* stream)
```

End of file can be manually raised when reading from a stream using `CTRL + D`.

fopen

First we upgrade our file descriptor to a file pointer. A pointer to a `FILE` struct.

This file pointer points to a stream, and may be created using the `fopen` function.

```
FILE* file_pointer = fopen("turtles.txt", "w");
```

Here

- `file_pointer` is a file pointer variable, if fopen has failed then it will have a value of NULL, otherwise it will point to the file.
- `"turtles.txt"` is the path to our file
- `"w"` is the mode with which we are opening the file

fopen

Our modes here are analogous to the modes we saw with the raw `open` syscall. The `O_CREAT` flag is implicit in all writing and appending modes.

- `r` read - opens file for reading
- `w` write - opens file for writing from the beginning of the file
- `a` append - opens file for writing from the end of the file
- `rb` read binary - opens a binary file for reading
- `wb` write binary - write to a binary file
- `ab` append binary - append to a binary file
- `r+` open file for reading and writing
- `w+` open file for reading and writing
- `a+` open file for appending and writing

It should be emphasised that all write modes truncate the file to 0 length before writing, while append modes do not.

fclose

And of course, after opening our file, we should always close it.

```
fclose(file_pointer);
```

Default File Pointers

We have three default file pointers included by `stdio.h`:

- stdin - File pointer for standard input
- stdout - File pointer for standard output
- stderr - File pointer for starndard error

fread & fwrite

We can read binary data from a file pointer using the `fread` and `fwrite` commands.

```
fread(void* buf, size_t size, size_t n_memb, FILE* stream);
```

Here we are reading `n_memb` elements of size `size` from `stream` to `buf`.

```
fwrite(void* buf, size_t size, size_t n_memb, FILE* stream);
```

And here we are writing `n_memb` elements of size `size` from `buf` to `stream`.

We could use this to read and write characters by setting `size` to `sizeof(char)` and then setting `n_memb` to the number of characters.

fseek

Just as with our previous file descriptors, our file pointer comes with an `fseek`. function.

It works in a near identical fashion:

```
int fseek(FILE* stream, long offset, int whence)
```

With a slight semantic difference, our *file offset* is now a *file position indicator*, as we are working with a stream, not a file descriptor.

fseek

With our new streams, comes some new functionality.

```
long ftell(FILE* stream);
```

Returns the current value of our file position indicator.

We also have a few helper functions that perform common seek operations in a more readable fashion. For example:

```
void rewind(FILE* stream);
```

Resets a file position indicator to the start of the file. which is identical to `fseek(stream, 0, SET_SEEK);`

Line IO

There exist many common patterns in reading and writing C strings.

- Reading till a newline character
- Appending null terminators to the end of strings
- Writing till a null terminator

This could be implemented using `fread` and `fwrite`, a switch and a loop.

However `stdio` also provides a number of functions that provide this utility for us.

gets & puts

`gets` and `puts` are our first line IO functions, they analogues of `read` and `write`.

```
char* gets(char* buf);
```

Simply reads from standard input to `buf` until a newline or `EOF` is reached. On error a `NULL` is returned, otherwise `buf` is returned.

This function has a very obvious problem; what if the buffer is smaller than the input?

It simply begins running through the rest of memory!

For this reason, **NEVER USE** `gets`.

gets & puts

`puts` is much more sensible. It simply writes from a buffer to standard output until it reaches a terminating null byte.

```
int puts(char* buf);
```

Puts returns a non-negative number on success, or EOF on error.

fgets & fputs

`fgets` and `fputs` are more general implementations of `gets` and `puts` and accept file pointers.

We could pass these pointers as `stdin` and `stdout` to replicate the behaviour of `puts` and `gets`.

```
char* fgets(char* buf, int size, FILE* stream);
```

Unlike gets, `fgets` only reads up to `size-1` bytes to but then appends a null terminator character. `fgets` will also break on newline characters and EOF, when an EOF is reached, fgets will return NULL.

fgets & fputs

`fputs` is very similar to `puts`, with the addition of the file pointer.

```
int fputs(char* buf, FILE* stream);
```

`fputs` returns a non-negative number on success, or EOF on error.

Formatted IO

While we can now read and write to and from files and streams, we still have some issues with converting variables to string representations and strings to variables.

We could carve out types from selected sections of the buffer as required ourselves, or resort to non-ANSI compliant C functions (itoa).

However, it's more useful to have a generic, general interface for managing our type conversions. For this we have our format strings you should be familiar with from `printf`.

fprintf

`fprintf` is a close relative of the `printf` function with which you are familiar.

```
int fprintf(FILE* stream, const char *format, ...)
```

Prints a string to a file pointer `stream` using the format string and trailing variables.

By setting the file pointer to stdout we replicate the behaviour of `printf`.

fscanf

`fscanf` is the `write` pair to `fprintf`, and is used to write to a file pointer.

```
int fscanf(FILE* stream, const char *format, ...)
```

`fscanf` reads from `stream` and formats the types based on `format` before assigning the values to the addresses passed as the variadic arguments. The format string syntax is shared between the scanf family and the printf family.

```
int x = 0;     fscanf(stdin, "%d", &x)
```

There also exists a `scanf` analogue to `printf` with `stdin` set as the file pointer.

Formatted String Manipulation

These string format methods are incredibly flexible, and as a result my also be used on regular strings rather than just file streams.

```
sprintf(const char* str, const char* format, ...)
sscanff(const char* str, const char* format, ...)
```

read write

Uses the read and write syscalls, then performs the conversion without a format string.

```
char buffer[BUF_SIZE];
while (read(fd_in, buffer, BUF_SIZE - 1)) {
    int num = atoi(buffer);
    char print_buffer[BUF_SIZE];
    itoa(num, print_buffer, 10);
    // ^ non-ANSI compliant function!
    write(fd_out, print_buffer, strlen(print_buffer));
}
```

feof fscanf fprintf

A simple approach to scan until the end of file.

```
while(!feof(stream)) {
    int num;
    fscanf(stream, "%d", &num);
    fprintf(stdout, "num: %d\n", num);
}
```

fgets sscanf fprintf

A more flexible approach that reads lines then formats them.

```
char buffer[BUF_SIZE];
while (NULL != fgets(stream)) {
    int num;
    sscanf(buffer, "%d", &num);
    fprintf(stdout, "num: %d\n", num);
}
```

Summary

The underlying file system is built upon abstractions

Low level I/O deals with system level objects and requires system calls to do so. These are resources and need to be managed by OS.

C level I/O deals with further abstractions of a file into a stream and supports various modes

Data can be extracted from a selected part of a buffer, or a stream and manipulated to store into the datatype of interest.

Write your own `itoa()`!