INFO1910 Week 6 Tutorial

IO

File Descriptors

In UNIX, all input and output is performed by reading or writing to files. All peripheral devices (keyboards, mice, screens...), are simply files that can be read from or written to. Processes may also be written to and read from. This means that a single homogeneous interface handles all communication between programs, between programs and devices and for reading and writing to data storage (files).

The first and most basic case is opening a file, the system checks your right to do so (permissions, existence checks etc). If these checks are passed, the program is returned an integer termed the file descriptor, the descriptor returned depends on whether read or write permissions are requested, and is used to perform these actions on the file.

In C the `open` function opens a file, and returns the associated file descriptor. It takes a path to the file, and flags specifying the mode it is to be opened in, for instance `O_RDONLY` for read only. These flags can be composed using the bitwise OR operator.

```c
int open(const char* path, int oflag, ...);
```

When the shell runs a program, three files are opened with descriptors 0, 1 and 2 termed standard input, standard output and standard error. The output of the keyboard is passed to the shell, which is then written to the file descriptor for standard input for the program, when the program prints values to standard output these are written to the associated file descriptor, the shell program can then read from this descriptor and display the text.

If we can open a file, we must also be able to close it.

```c
int close(int fd)
```

The read and write functions can be used on a file descriptor (that has been opened with the appropriate flags), to write from or read a fixed number of bytes to a buffer.

```c
int n_read = read(int fd, char* buf, int n_bytes);
int n_written = write(int fd, char* buf, int n_bytes);
```

It's also important to note that the buffer that we are writing to (int the case of the read function) must be at least `n_bytes + 1` bytes long. The excess byte is then used to store our null terminator.

Question 1: Flag Composition

Explain how the bitwise OR operator can be used to compose flags for the open function. Printing some flags might assist with this.

Question 2: Read and Write

Set up a 128 byte buffer, and use the read and write functions on stdin (fd 0) and stdout (fd 1) to read a string from standard input, and print to standard output.

Defining a `BUFFER_SIZE` identifier provides a decent consistent buffer size without worrying about magic values. Be aware of the need to keep a byte for the null terminator for your strings.

lseek

The lseek sys call allows us to offset our position in the current file given by the file descriptor. This allows us to move back and forward within the file without needing to open the file again.

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

Here the arguments are the file descriptor,

Whence can take a number of arguments, these are `SEEK_SET`, which sets the offset to offset bytes, `SEEK_CUR` which sets it to its current location plus offset bytes, and `SEEK_END` which sets it to the end of the file plus offset bytes.

Question 3: File Length

Write a C program that takes a file path single command line argument and prints the number of bytes in that file. If the file does not exist, you should print `File does not exist!`.

- Do this using the `read` function
- Do this using `seek`.

Don't forget to set your flags appropriately when opening the file.

Fopen

While `open` returns a file descriptor, `fopen` returns a file pointer. This is a higher level abstraction of our file descriptor and offers a bit more support.

Here, rather than composing our mode from the previous flags, the mode is passed as a string, `FILE* fopen(const char* path, const char* mode)`

With our new file pointers, we can now use functions that deal with these objects rather than the descriptors themselves.

Gets and Puts

NEVER USE THE GETS FUNCTION We'll start this section with looking at the man page for gets. Having observed the man page for gets you can now forget about it; it is only to be included here for historical purposes, it should never be used.

One step removed from our basic read and write functions we have puts and gets. These were developed with the idea that you would no longer need to specify how many bytes were being passed. Shortly thereafter it was decided that this was a terrible idea for reading input and was condemned.

Starting with getc and fgetc, we can get a single character from a file stream. `stdin` is a file stream object. (file get char).

```
int fgetc(FILE* stream);
```

By iterating this call and saving the output to a buffer, we reach the fgets function (file get string). Fgets terminates on the end of file (EOF), a new line, or when one less than size bytes of input have been read. The function will then insert a null terminator after the last byte of the string.

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

Upon receiving an EOF (Ctrl + D), fgets will return a NULL. A useful method of accepting arbitrary input is then to loop over fgets until a null is printed.

```
while (NULL != fgets(buffer, size, stdin))
{
    // Stuff
}
```

Similarly we see the puts and fputs commands. These print to a given file stream.

```
int fputc(int c, FILE* stream);
int fputs(const char* s, FILE* stream);
```

The `puts` function itself just substitutes the file stream for stdout and appends trailing newline character.

Question 4: Summation

Write a program that reads from standard input until it receives an EOF. It should interpret each input as an integer when it receives the EOF it should print the sum of all the numbers.

Printf, Scanf, Format Strings

We have previously seen `printf` and its format strings, and you will have seen `scanf` in previous lectures. The main advantage of these functions over puts and gets is the inclusion of a format string.

With the addition of the format string, the arguments to the functions are mapped to their string representations prior to printing, whereas previously this task had to be performed manually. Similarly scanf will interpret strings as their appropriate type using its own format string argument.

While you have encountered printf and scanf before, we will introduce file printf `fprintf` and string printf `sprintf`. These print to files and string respectively.

Similarly we get `sscanf and fscanf` to read from existing strings and files.

Question 5: Fibonacci Returns

Write a program that takes an integer command line argument `n` then prints the first `n` Fibonnaci numbers to a file. You should implement this iteratively.

Putting it all Together

We can make use of the best of both of these approaches to first read from standard input, then format the strings. This uses a combination of fgets followed by sscanf.

```
while (NULL != fgets(buffer, size, stdin))
{
    int x, y;
    sscanf(buffer, "%d %d", &x, &y);

    printf("%d \n", x + y);
}
```

Here, we can also replace our stdin with a file pointer returned by the `fopen` function.

Question 6: File Read

Write a program that reads your Fibonacci file and finds the sum of all the numbers therein.

Pipe or redirect your the Fibonnacci file to your previous summation program and see if you get the same result.