THE UNIVERSITY OF SYDNEY

INFO1910 S2 2023 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.

```
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.

```
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.

```
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.

```
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. Along with this file stream object we also have the `fread` and `fwrite` library calls.

Question 4: Display

What is the output of each of these writes.

```c
int main()
{
    char str[20] = "Hello world!\n";

    int arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    fwrite(str, sizeof(char), 13, stdout);
    fwrite(arr, sizeof(int), 10, stdout);

    return 0;
}
```

Question 5: Buffering

Calling read and write is a time intensive process. Calling it for every character leads to large overheads. ‘read’ and ‘write’ are performed character by character.

Conversely our file stream objects are buffered, that is there is an internal userland buffer that is written to by our file pointers, which is occasionally flushed to the pipe. Conversely the write syscall writes to the pipe character by character and is non-buffered.

Predict what the following code will do then run it. Did your prediction match what occurred?

```c
int main()
{
    char* ptr = NULL;

    fprintf(stdout, "Hello world!");
    // fflush(stdout);
    write(1, "Write ahoy", 10);
```

```
    ptr[0] = 'a';
    return 0;
}
```

Uncomment the flush to force userland buffer to flush to the pipe. What ascii character normally forces the buffer to flush?

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 6: 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.

Question 7: atoi

Write your own implementation of the atoi function. Your function should take a string as input and return the integer representation of the contents of the string.

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 8: 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 9: 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.

Question 10: Reverse Polish

Reverse polish notation is a different way of writing mathematical expressions where the operator occurs after the operands. For instance $3 + 4$ would become $34+$. This particular representation doesn't require bracketing to resolve breaks with standard orders of operations, however it is generally less human readable.

We will start with pairwise operators on each line, the `%` symbol indicates that this value should be the output of the previous line. You will have to implement the ADD, SUB, MUL and DIV operators.

Write a program that reads lines of reverse polish notation and prints the current state on each line, and the final state after an EOF has been sent.

```
5 6 ADD
% 3 MUL
```

Your reverse polish code should be stored in an `.rp` file. The file name should be passed to your program as an argument. You might find function pointers useful here.

Question 11: Python

Suggest how a python interpreter might work given your previous implementation of the reverse polish calculator.

As an extension, write your own Python interpreter in C using the `fork` and `exec` commands (which we will discuss later). You should accept a file as a command line argument then pass the contents of the file to Python. Start with:

```
print("Hello World");
```

Fseek

As we've already seen lseek acting on the file descriptors, we also get fseek to act on our file pointers. Most of the arguments are shared, though there are a few helper functions.

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

Where whence has our `SEEK_SET SEEK_CUR SEEK_END` options from lseek. We also get a rewind function that is equivalent to `fseek(stream, 0, SEEK_SET)`, but a bit more verbose. Otherwise this acts in the same manner as our lseek.

Question 12: Progress Bar

Write a progress bar, you will need the `fflush` function to force the buffer to print without a newline character, and either `fseek` or `lseek` the `\b` character might also be helpful..

Question 13: Extension: Reorder

Write some text to a file, then write a program that will open the file. Give every character in the file a value based on its position. Starting at the last position in the file, You are to print the contents of the file as follows:

- If the current position is even, divide it by two
- If the current position is odd, triple it and add one

If this would result in reading past the end of the file, stop printing characters until your position returns to a point within the file.

Do this first using `open` and `lseek`, then using `fopen` and `fseek`.

Pipes

The underlying mechanism behind I/O is a 'pipe'. This is a dynamic memory buffer that is mapped to the address space of multiple processes. Some processes may write to the buffer, others may read from it. Once read from the default behaviour is to 'flush' the buffer which clears it for re-use. These pipes themselves otherwise act as files and have their own user and group IDs based on the calling process.

Pipes are a low level construct and act on file descriptors directly rather than on the file pointer objects. We'll look at the `pipe` and `fcntl`

`pipe` Opens a new pipe object and attaches it to two new file descriptors; `filedes[0]` is the output while `filedes[1]` is the input.

```
int file_des[2];
pipe(file_des);
write(file_des[1], "Hello!", 6);
read(file_des[0], buffer, 6);
```

When passing from the terminal to stdin this is also buffered and the buffer is only flushed and pushed to the appropriate pipe when a newline character is passed.

Question 14: Piping

Write a program that redirects standard input to standard output using an intermediary pipe and two buffers.

Non-Blocking IO

Normally when waiting for input the program will halt and wait for the input to be provided, however this is not always the desired behaviour. It stands to reason that if the memory for the pipe is mapped to somewhere in the current process then it may be probed to check if it contains data or not.

To achieve this we can use the `O_NONBLOCK` flag. If the flag is not already set for a particular file descriptor it can be managed using the `F_SETFL` option to the `fcntl` function. You will want to read some `man` pages.

Question 15: While we Wait

Write a program that increments an integer. In between increments it should check if stdin contains any data, if it does then the contents of stdin should be printed along with the current state of the integer. Compare how the program performs without the `O_NONBLOCK` flag.

Fork and Exec

These pipes are very interesting, but are so far useless when confined to a single process. The true power of pipes comes when they can be shared between processes, however to do this we must first consider how to invoke a new process.

A process can be roughly described as the current state of the registers (the context) along with the current state of memory mapped to that process (for example the stack). We can halt a process and resume it simply by saving the registers and restoring them.

We can similarly duplicate a process just by cloning its virtual address space and its context. The fork command performs this trick. The only distinction between the original 'parent' process and the 'child' process is that the fork function returns 0 for the child.

Exec similarly remounts the program code space with data from a new file and resets the registers to execute from the entry point. This replace an existing process with another one.

Question 16: Pipes Again

Write a program that forks and then:

- The child calculates prime numbers up to 1000 and pipes them to the parent
- The parent should perform a non-blocking read on the pipe and redirect the result to stdout.

Don't forget to close the ends of the pipe that aren't in use!

Question 17: Extension: Reverse Polish 2

Extend your reverse polish representation to allow for lines of up to 128 characters containing a variable number of reverse polish operations. You will now need to deal with parsing more complex statements.

```
5 6 SUB 7 ADD 8 MUL
% % ADD % DIV
```

Question 18: Extension: itoa

Write a function itoa that takes an integer and a buffer and fills the buffer with the string representation of the integer.

Question 19: Extension: Terminal

Write a program that emulates a very simple shell.

- stdin is to be interpreted as a call for forking and execing a particular file.
- Child processes that are spawned in such a way should have their standard input and output redirected to the parent processes' standard inputs and outputs
- standard input that ends with the `&` character should be interpreted to background the process and not forward standard input to it.