CSE333 Exercise 4

Out: Wednesday, January 19
Due: Wednesday, January 26 by **11 am PST**
Rating: 5 (note)

Goals

- Use various POSIX I/O library functions with files and directories
- Implement a buffered system
- Recognize the relationship between C standard library I/O and POSIX I/O
- Implement code that error checks I/O function calls and properly cleans up resources on every execution path

Background

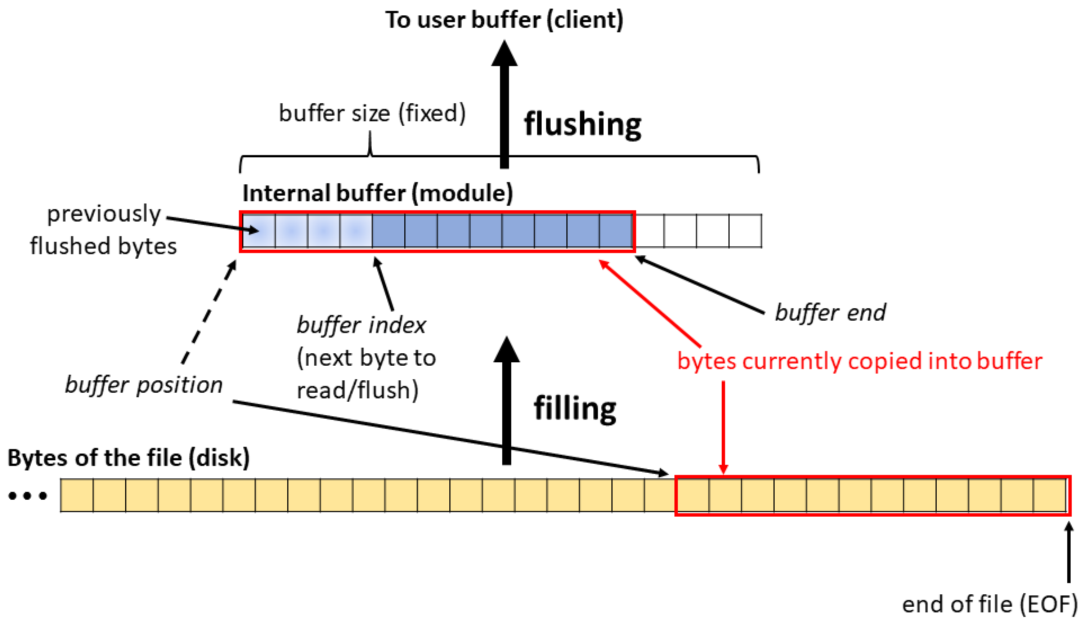
Directories

Directories are special files that store the names and locations of the related files/directories — itself, its parent, and all of its children (*i.e.*, the directory's contents). You can take CSE 451 to learn more about the directory structure, but for this class, you can access directory information in C/C++ using the `struct dirent` structure and the POSIX library functions found in `dirent.h`.

Using directories is not covered in lecture but is discussed in Section 3 along with an example program. You may refer to these slides (linked once section materials are) for more information on how to read directories in C (the slides have animations so click "Present" to see them).

Buffering

As discussed in lecture, C standard library I/O is implemented using system calls (POSIX) with an internal buffer to optimize performance. In this exercise, we will focus on *buffered reading*, where chunks of data from the file are stored in an internal buffer so that requests from the user can be returned more quickly from the internal buffer in memory rather than making a request to the file on disk each time. We must manage the internal buffer properly so that it is invisible to the client:



(buffering.png)

- Because we don't know the size of the file ahead of time and reading the entire file contents at once would defeat the purpose of buffering (and be very slow for large files), we will use a ***fixed size buffer***. This acts as an upper bound on the size of the "window" of file data we can store at one time, with a "**buffer position**" indicating where this window starts in the file and a "**buffer end**" indicating where this window ends in the buffer.
- As we send bytes from our buffer to the user, we don't want to shift data in the buffer, as this is an expensive linear-time array operation involving memory accesses. Instead, we will move an indicator within our buffer (a "**buffer index**") to keep track of our current position within the buffer.
- You should consider how these buffer indicators change for the following common operations:
  - Filling the buffer – reading data from the file into the internal buffer, overwriting previous data
  - Flushing the buffer – sending valid file data from the buffer to the user
  - Seeking within the file – changing file positions without reading or writing

Problem Description

This exercise will be broken into multiple parts:

1. You will write a program that prints out the contents of all text files in a directory to `stdout` using a combination of C standard library and POSIX library functions.
2. Then you will complete an implementation of a file I/O program that mimics what is done by some of the C standard library I/O functions using only POSIX library functions.
3. Then you will revise your original directory program to replace the C standard library file I/O functions with your homegrown ones!

Provided Files

We have provided you with the following **seven** source files, which can be downloaded from this web directory (./ex4_files) or with the commands:

```
ash$ wget https://courses.cs.washington.edu/courses/cse333/22wi/exercises/ex4_files/<filename>
```

- **`ex4.c`** — Contains an empty `main` for the directory program, but does have helper functions to handle the major C-string operations.
- **`ro_file.h`** — Provides the public interface for our file I/O module, which includes a file management information data type and file I/O operations. Note that `struct ro_file_st` is defined in `ro_file.c`.
- **`ro_file.c`** — Implementation file for our file I/O module that you will need to complete.
- **`test_ro_file.c`** — Implements an executable that makes use of the file I/O operations provided by `ro_file.h`.
- **`test.txt`** and **`test2.txt`** — Two text files used in `test_ro_file.c`.
- **`Makefile`** — Provided for your convenience in compiling the executables `ex4` and `test_ro_file`.

Note: You will only submit `ex4.c` and `ro_file.c`.

Step 1: Write the directory program

Write `main` in `ex4.c`. Your program should:

- Accept a directory name as a command-line argument. The directory name can be a simple name or a longer file path and might, or might not, have a trailing `'/'` at the end.
- Scan through the directory looking for filenames that end in the four characters "**`.txt`**". You do not need to scan subdirectories recursively.
- For each text file, read the contents using *C standard library functions* and write them to `stdout`. Do NOT add any additional characters or formatting (*e.g.*, don't add newlines between the contents of different files)!

We have not provided any tests for you in this part, but we encourage you to test it on various inputs.

Step 2: Implement the file I/O module

1. Read the background information about buffering, carefully examine the `struct ro_file_st` definition in `ro_file.c`, and then read the function comments in `ro_file.h`.
2. Complete the functions found in `ro_file.c`.
3. Run the tests in `test_ro_file.c`.
    - These tests are not exhaustive and passing them does not guarantee a full correctness score. We encourage you to add additional tests here, however, these are optional and won't be submitted. In particular, you should test files that are bigger than the size of the internal buffer.
    - The provided tests do not make sure that you use the internal buffer but, when grading, we will be checking that you do so.

Step 3: Use the file I/O module in the directory program

Now go back and revise `ex4.c` to replace the C standard library functions for file reading with calls to the file I/O module. You are allowed to keep using C standard library functions to write to `stdout`.

Recommendations:

- Keep a copy of your working `ex4.c` from Step 1 somewhere for partial credit opportunities in the event that you don't finish debugging `ro_file.c`.
- Re-run any tests you used in Step 1 (and add more?) to verify the behavior of your file I/O module.

Implementation Notes

User Input

You will be reading in user input from a command-line argument. You should handle various inputs from the user, which may be in an unexpected format. For each input, you should take some time to reason through your options for handling it gracefully (*i.e.*, without unexpectedly crashing), decide which one seems "best", and document your decision in your code. Refer to the Step 1 instuctions above for some directory name considerations.

Testing

There are various levels of testing to be done in this exercise, including (1) user input testing, (2) input file testing, and (3) testing individual functions that you've written (*i.e.*, debugging). For input file testing, you are encouraged to write your own files (`.txt` and otherwise) but can also find plenty of online samples (http://textfiles.com/directory.html).

POSIX

We are restricting ourselves to only POSIX library functions for finishing our `ro_file` implementation; you are NOT allowed to use functions from `cstdio`. The POSIX functions you will likely be most interested in are: `open()`, `close()`, `read()`, `lseek()`, `opendir()`, `readdir()`, and `closedir()`.

Error Handling & Robustness

Library functions (*e.g.*, C standard library and POSIX) have many possible errors that may arise during execution. It is your responsibility to make sure that you handle these errors correctly by retrying in the case of recoverable errors (`EAGAIN` and `EINTR`) and returning an error status in the case of a non-recoverable error. Make sure that you clean up system resources in *all* possible cases.

Style Focus

Don't forget that all of the good practices from previous exercises still apply!

General

For the sake of our autograder, **you may not modify `ro_file.h`**. You are free to modify any other files as you see fit for design and/or testing purposes, but recall that you will only be submitting `ex4.c` and `ro_file.c`. If you add internal data types, constants, or functions to `ro_file.c`, you should document your changes and explain why you made them.

Style Consistency

Since you have been provided code, it is important to match the given style for consistency, including commenting style, indentation, data types (*e.g.*, `size_t`, `ssize_t`, `off_t`), and error checking.

In addition, pay particular attention to comments, as some have been provided to help you plan your implementation but may or may not make sense to keep in your submitted version. Remember that comments should be up-to-date and meaningful; don't assume that all provided comments can be left as-is.

Submission

You will submit: `ex4.c` and `ro_file.c`.

Your code must:

- Compile without errors or warnings on CSE Linux machines (lab workstations, `attu`, or CSE home VM).
- Have no runtime errors, memory leaks, or memory errors (`gcc` and `valgrind`).
- Compile with the original, provided `Makefile`.
- Have a comment at the top of your submitted `.c` files with your name(s) and CSE or UW email address(es).
- Be pretty: the formatting, modularization, variable and function names, commenting, and so on should be consistent with class style guidelines. Additionally, the linter shouldn't have any complaints about your code (`clint.py`).
- Be robust: your code should deal with hard-to-handle/edge cases and bogus user input (if there are any) gracefully.

Submit your code on Gradescope (https://www.gradescope.com). Don't forget to add your partner if you have one.

PAUL G. ALLEN SCHOOL OF COMPUTER SCIENCE & ENGINEERING

UW Site Use Agreement (//www.washington.edu/online/terms/)