Multi-threaded Word Wrap

(Combined Project 2 & 3 Documentation)
CS [redacted] - Prof. [redacted]
Authors: [redacted]
Group [redacted]
ReadMe and Testing Strategies

C Implementation

Files

ww.c
- Required word wrapping functionalities, file IO, and handling of threads.

queue.h
- All functions relating to queues.

Makefile
- Tags used for checking memory errors, space leaks, and undefined behaviors:

Variables, Structs, and Functions

ww.c

pthread_mutex_t access_file_q, access_directory_queue
- Mutex locks to prevent multiple threads from accessing the queue at the same time.

Queue_node DirQueue, FileQueue
- Queues for storing directories to be read and files to be wrapped

int active_threads
- The number of threads actively reading/wrapping.

struct handleFile_arg
- int wrap
  - Passing linesize limit
- int* ERRFLAGS
  - Passing the error flag variable address
- Struct to pass arguments to handleFile with pthread_create

void printwrap(int fd_src, int fd_dest, int lineSize, int* ERRFLAGS)
- This function is the main implementation for wrapping. The function accepts the input as “fd_src,” output as “fd_dest,” and the desired width of the line as “lineSize.” It

additionally checks whether a line had to be printed over the wrapping limit triggering an EXIT_FAILURE by setting ERRFLAGS to 1.

void *handleDir(void* unused)

- This function handles the scheduling of threads and reading of directories. The function transverses through every directory on the queue. It automatically adds new directories and files as they are found. The argument passed is unused

void *handleFile(void* arg)

- This function handles the scheduling of threads and calls to printwrap. It will pop every file off the file queue and wrap them. To be called with linesize and error flag address stored in arg.

int check_name(char* name)

- Helper function to check whether a file name is valid or not to add to the queue.

queue.h

struct queue_node

- char* file_path
  - Stores the file path of the current directory/file
- struct queue_node* next;
  - Stores the next node address
- Basic queue node

queue_node* init_queue(queue_node* start)

- Initializes the queue. Must be called before any other queue functions.

int is_empty(queue_node* start)

- Returns status of queue size. Returns 1 if empty otherwise returns 0.

char* peek(queue_node* start)

- Returns the first nodes file path. Equivalent to calling start->file_path.

int queue_len(queue_node* start)

- Returns size of queue.

char* pop_node(queue_node* start)

- Deletes the first node and returns its file_path.

queue_node* insert_node(queue_node* start, char* txt, int len)

- Inserts a node at the end of the queue provided and copies the data from txt up to len bytes.

queue_node* clear_queue(queue_node* start)

- Deletes every node in the queue.
    - Implemented via successive calls to pop until is_empty returns true.

Implementation

Our current implementation of the program relies on calling the printwrap() function for all different cases, where the number of command line arguments used to run the program helps determine the input (where printwrap() reads data from) and the output (where printwrap() writes data to). The different cases are handled in the following manner:

1. If there are less than 2 arguments: Return EXIT_FAILURE.
2. If there are exactly two arguments: Call printwrap() with input as stdin and output is stdout
3. If the third input is a valid file name: Call printwrap() with input as the file_descriptor for the file and output is stdout
4. If the third input is a valid directory name: Open the directory and for each valid file in the directory, create an output file with the prefix 'wrap.' and call printwrap() with input as each file and output as the corresponding output file.
5. If the third input is invalid: Call printwrap() with input as stdin and output is stdout

The function uses various variables to test conditions while printing the formatted words. chars_printed keeps track of the words printed to screen. Buffer stores any words written to the buffer before being printed. Newline keeps track of the amount of newlines we have printed. Chk stores read() returns to check for end of file. Buf keeps track of whether we had something stored in the buffer on new lines. The function first attempts to print the first word until EOF or it encounters a space. Then it will use the buffer to print subsequent words by checking their lengths before printing them. If necessary printing on the next line.

Edge case: In case of a word longer than the desired width, the word is printed on a newline by itself.

Additionally, program has been expanded to be able to read through directories recursively. The first argument provided may optionally be a "-r" denoting that any directories are to be traveled recursively. Any files and directories found are added to a queue. The function handleDir automatically pops and processes every item off the queue. The function also has the ability to be called by multiple threads so that directories can be read concurrently. The user specifies this via number arguments right after -r. Similarly the function handleFile is called after to perform concurrent file wrapping through a similar process.

The program also iterates through every argument provided, attempting to wrap or read directories depending on the argument given.

Testing

Upon careful consideration, we determined that the correctness of the program depends on the following:

1. Handling different numbers of arguments.

- The program was run multiple times to check all possible cases like (but not limited to):
    - with and without '-r' argument
    - with different widths
    - with a single file path
    - with a single directory path
    - with directory path and file path (extra credit)
- Multiple printf() statements were used throughout the code during the debugging phase to check for the file handling logic. They have been removed for submission.

2. Wrapping exactly within the correct lengths.

- The code was first tested with less data and a small lineSize to be able to manually check for wrapping boundaries and errors by writing the output into a text file. This helped correct a lot of edge cases that we first didn't account for.

3. Removal of unnecessary empty spaces.

- The code was separately tested against multiple files with unnecessary empty spaces in various different positions of the file. The code behaved unnaturally with empty spaces at the end of file. This error was further rectified.

4. Normalization of the output

- All the tests mentioned above as well as the tests done after were done by writing the output into text files, therefore the code was run twice for each test input to check for normalization while checking for other errors.

5. Handling edge cases:
    a. Empty file: write a newline.
    b. Empty directory or no readable files: Return EXIT_FAILURE
    c. Words larger than lineSize: Print the word on a newline and return EXIT_FAILURE
    d. lineSize of 1: Print all words on a newline

6. Handling large files.

- The code was tested against input files copied from a combination of 3 to 4 book pages at a time found on the internet.

7. Multithreading is being used as requested in the arguments.

- This was tested by maintaining and printing a variable that keeps track of active threads at any given time during the tests. This helped us confirm that not only multiple threads are being used, but also that the count of active threads doesn't exceed the requested number.

8. The directories are being explored recursively only when the argument '-r' is used..

- This was tested by running the program with and without the argument '-r' on a directory that has subdirectories within. In the case where '-r' was a given argument, the program explores the given directory and all its subdirectories. On the other hand, without the '-r' argument, the code would only wrap the files immediately in the given directory and ignore the subdirectories.

Compilation

To build the program, please ensure that you are in the working directory in the terminal and then run the following code:
make clean
make

To run the program:
Replace <line limit> with the required length of line. (integer)
Replace <file name> with a valid path to file
Replace <directory name> with a valid path to the directory.
Replace <recursive qualifier> with -r, -rN, -rM,N (where N and M are integers representing maximum count of reading and wrapping threads respectively)
Replace … with additional <file name> or <directory name> paths

1. To word wrap using the standard input:
    a. If provided more than one path it will create new files instead of outputting to standard output.

   ./ww <line limit> …
2. To word wrap the contents of a single file and get its output in standard output:
   ./ww <line limit> <file path>
3. To word wrap the contents of all immediate files in a directory:
   ./ww <line limit> <directory path> …
4. To word wrap the contents of all immediate files in directories and some files:
   ./ww <line limit> <directory path> …
5. To word wrap the contents of files in a directory and its subdirectories:
   ./ww <recursive qualifier> <line limit> <directory path> …
6. To word wrap the contents of files in a directory and its subdirectories and some files:
   ./ww <recursive qualifier> <line limit> <directory path> …