Multithreaded Word Wrap

Purpose

The purpose of this program is to demonstrate the producer-consumer model using Posix threads. The producer threads **concurrently** enqueue directory and sub-directory paths in the unbounded directory stack and also enqueue regular file paths in the bounded file queue. The consumer threads also **concurrently** dequeue the regular file paths from the file queue, wrapping the content from the regular file and finally writing the wrapped content to a new regular file in the same directory as the regular input file. The new file can be identified by `wrap.input_file_name.txt`.

Unbounded Directory Stack or Pool

Data Structure Purpose

The pool itself is a linear LIFO (Last In First Out) data structure that has the capability to expand once the pool is full. It is used as directory stack in our project.

Pool Model

```
 struct Pool_Data
{
    char *directory_path;
};
struct Pool_Model
{
    pool_data_type** data;
    int end;
    int pool_size;
    pthread_mutex_t lock;
    pthread_cond_t ready_to_consume;
    int number_of_elements_buffered;
    int number_of_active_producers;
    bool close;
};
```

1. The `pool_data_type** data` field is the actual unbounded directory stack or pool where every element in the pool is a pointer to the `pool_data_type` struct. The `pool_data_type` struct consist of only a char pointer that will store the directory or sub directories path.
2. The `end` field is the end pointer that is incremented when an element in enqueued to the pool and decremented when an element is dequeued. Initially it is set to 0.
3. The `pool_size` field stores the inital capacity of Pool. The `pool_size` is doubled everytime the pool is full of data, thus the unbounded nature of the directory stack. The initial `pool_size` is stored as a macro called `POOLSIZE` in the `word_break.h` file.
4. The `lock` field is a mutex lock of type `pthread_mutex_t`. Only one lock is initialized when the pool is initialized and destroyed when the pool is destroyed. This lock makes our pool thread-safe meaning it protects all the fields of `Pool_Model` from thread races. For more details, please look at the Mutex part in the Synchronization section.
5. The `ready_to_consume` is a condition variable. In short, it is used to block the dequeue method until enqueue method fills data in the `Pool`. For more details, please look at the Condition Variable part in the Synchronization section.
6. The `number_of_elements_buffered` field stores the number of elements present inside the pool at any given time. The `number_of_elements_buffered` field is incremented when an element in enqueued to the pool and decremented when an element is dequeued.
7. The `number_of_active_producers` field is developed to keep track of the number of directory threads that are reading directories at a given point of time. This count is incremented when a directory thread dequeues the directory path from the pool and starts reading the directory. The count is decremented after the directory thread has enqueued all sub-directories from the directory path and done reading the directory.
8. The `close` field is a boolean flag to indicate whether the pool will accept anymore data. If the flag is set to `True` then all of the remaining directory threads that are responsbile for enqueuing and dequeuing the directories from the directory stack are woken up using the `broadcast` signal method, and thus all the remaining directory threads are signaled to finish enqueuing or dequeuing the remaining data. `Note:` This field prevents the directory threads from getting hanged because the dequeue method will `not` wait/block until the enqueue fills data because the `close` field indicates that there is no data available for the enqueue method to enqueue data anymore.

Synchronization

Mutex

We used `one` mutex lock provided as a field in the `Stack` struct to read/write any field in the `Stack` struct in a exclusive way. The mutex lock provides exclusive access for the threads to read and write the stack internals (e.g. data, pool_size) one at a time in order to avoid multiple threads reading and writing the stack internals at once. So when a thread `t1` acquires the lock for reading or writing the stack internals, and if another thread `t2` attempts to read or write the stack internals at the same time then `t2` will sleep until `t1` releases the mutex lock.

Condition Variable

We also used `one` condition variable called `ready_to_consume` provided as a field in the `Stack` struct for inter thread communication purposes. `ready_to_consume` is set to wait in our `pool_dequeue` method when the the stack is empty. Once the `wait` is called in dequeue, the thread responsible for calling dequeue is set to sleep, the mutex lock is released temporarily and finally, the control is passed to enqueue. Enqueue then locks the mutex again, and enqueues data into the stack and `signals` the condition variable `ready_to_consume` so that dequeue won't be blocked anymore, and then dequeue can start dequeuing data once again because there is some data in the stack.

Unboundedness nature of the Stack

The reallocation of the stack happens in our `pool_enqueue` method. So when the stack is full, we `double` the size of the stack to `2n` where `n` is the original size of the stack. We also make sure to copy all of the existing data from the original stack of size `n` into the new stack of size `2n`.

Why use an unbounded stack instead of a unbounded circular queue?

One more thing to mention is that reallocation was the primary reason we used a stack rather than a circular queue. Since stack only has `one pointer` at the end, while a circular queue has `two pointers` one at the start (for dequeue), and one at the end (for enqueue), so it was trivial to reallocate the stack with one pointer without worrying about any edge cases.

A example of when a stack is useful than a circular queue. Lets say Circular Queue looks like this:

```
Circular_Queue = [ 1, 2, 3, 4, 5]
                            ^  ^
                            e  s
```

Assuming e = end pointer, and s = start pointer. If we reallocate Circular_Queue then we have to double the size of the Circular_Queue and re-order the data in the `new` double sized Circular_Queue so that `start` pointer comes first and the `end` pointer comes last. So the final result will look like [5, 1, 2, 3, 4, _, _, _, _, _, _]

```
Unbounded_Stack = [ 1, 2, 3, 4, 5]
                                ^
                                e
```

Assuming e = end pointer. If we reallocate Unbounded_Stack then we have to double the size of the Unbounded_Stack and just copy the data from index to 0 to the end pointer in the `new` double sized Unbounded_Stack. So the final result will look like [1, 2, 3, 4, 5, _, _, _, _, _, _]

Unit-Tests for Pool

Our unit test for Pool can be found in `unit_test/pool_test.c`. We tested the pool by two main testing approaches. 1. Vanilla Test (no threading) is meant to just enqueue certain number of elements, and dequeue the pool until its empty without the use of any type of threading. So if the pool is empty and there are no memory leaks then the Vanilla Test ran successfully. 2. Threading Test is designed to test out the producer-consumer model on the pool. We created a producer worker function that produces/enqueues `n` number of elements in the pool. We also created a consumer worker function that consumes/dequeues the elements from the pool until the pool is empty, and the producer has no more data to produce. Notice that unlike the vanilla test, this test will concurrently enqueue, and dequeue elements from the pool. This will also test if the pthread synchronization devices work as expected. So if the pool is empty after every test and there are no memory leaks then the threading Test ran successfully. 3. General conditions for the test. 1. For the threading test, we tried different combinations for the number of producer, and the number of consumer threads that will target the producer worker function, and consumer worker function respectivily. For instance let's say a pair of producer, and consumer threads is represented as `(p,c)`. Then we tested the threading test with the pairs (1,1), (3,5), (5,3), (4,4). 2. For the vanilla, and threading test we tested it with 1000 elements. This number can be changed in the main method of `unit_test/pool_test.c` 3. Since this is an unbounded stack, we wanted to also test the reallocation mechanism of our stack. So we ran vanilla and threading test with the initial pool size of 1, 10, 100, 1000. These initial sizes reallocated the stack when neccessary (see condition for reallocation above).

Bounded File Queue

Data Structure Purpose

The queue is a linear FIFO (First In First Out) data structure that is circular and bounded. It is used as a file queue in our project.

Queue Model

```
struct Queue_Data
{
    char *input_file;
    char *output_file;
};
typedef struct Queue_Data queue_data_type;

struct Queue_Model
{
    queue_data_type** data;
    size_t start;
    size_t end;
    size_t queue_size;
    pthread_mutex_t lock;
    pthread_cond_t ready_to_consume;
    pthread_cond_t ready_to_produce;
    size_t number_of_elements_buffered;
    bool close;
};
```

1. The `queue_data_type** data` field is the actual bounded file queue where every element in the queue is a pointer to the `queue_data_type` struct. The `queue_data_type` struct consist of a char pointer that will store the path of the input file that is to be wrapped and another char pointer that will store the output file where the wrapped content from the input file will be stored.
2. The `start` field is the start pointer that is always incremented when an element is dequeued from the queue. Since the queue is circular, if the start pointer is at the end of the queue `(Condition: start == queue_size)` then start will be reset back to the beginning of the queue `(start=0)` by doing `start % queue_size`. Initially `start` is set to 0.
3. The `end` field is the end pointer that is always incremented when an element in enqueued to the queue. Since the queue is circular, if the end pointer is at the end of the queue `(Condition: end == queue_size)` then start will be reset back to the beginning of the queue `(end=0)` by doing `end % queue_size`. Initially `end` is also set to 0.
4. The `queue_size` field stores the inital and fixed capacity of Queue. It is never changed during the lifetime of the Queue. The initial `queue_size` is stored as a macro called `QUEUESIZE` in the `word_break.h` file.
5. The `lock` field is a mutex lock of type `pthread_mutex_t`. Only one lock is initialized when the Queue is initialized and destroyed when the Queue is destroyed. This lock makes our Queue thread-safe meaning it protects all the fields of `Queue_Model` from thread races.
6. The `ready_to_consume` is a condition variable. In short, when the queue is empty, it is used to block the dequeue method until enqueue method fills data in the `Queue`. For more details, please look at the Condition Variable part in the Synchronization section.
7. The `ready_to_produce` is a condition variable. In short, when the queue is full, it is used to block the enqueue method until the dequeue method dequeues an element from the `Queue`. For more details, please look at the Condition Variable part in the Synchronization section.
8. The `number_of_elements_buffered` field stores the number of elements present inside the queue at any given time. The `number_of_elements_buffered` is incremented when an element in enqueued to the queue and decremented when an element is dequeued.
9. The `close` field is a boolean flag to indicate whether the queue will accept anymore data. If the flag is set to `True` then all of the remaining file threads/consumer threads that are responsbile for dequeuing the file paths from the file queue and wrapping them to their respective output file path are woken up using the `broadcast` signal method. Thus, all the remaining file threads are signaled to finish dequeuing and wrapping the remaining file paths from the file queue. `Note:` This field prevents the file threads from finishing early because the directory threads will only set this flag to true when all of the directories and sub-directories have been read and all the files from those directories and sub-directories have been enqueued into the file queue.

Synchronization

Mutex

We used `one` mutex lock provided as a field in the `Queue` struct to read/write any field in the `Queue` struct in a exclusive way. The mutex lock provides exclusive access for the threads to read and write the stack internals (e.g. data, queue_size) one at a time in order to avoid multiple threads reading and writing the stack internals at once. So when a thread `t1` acquires the lock for reading or writing the stack internals, and if another thread `t2` attempts to read or write the stack internals at the same time then `t2` will sleep until `t1` releases the mutex lock.

Condition Variable

We also used `two` condition variable called `ready_to_consume` and `ready_to_produce` provided as a field in the `Queue` struct for inter thread communication purposes. `ready_to_consume` is set to wait in our `queue_dequeue` method when the the stack is empty. Once the `wait` is called in dequeue, the thread responsible for calling dequeue is set to sleep, the mutex lock is released temporarily and finally, the control is passed to enqueue. Enqueue then locks the mutex again, and enqueues data into the stack and `signals` the condition variable `ready_to_consume` so that dequeue won't be blocked anymore, and then dequeue can start dequeuing data once again because there is some data in the stack. Similary `ready_to_produce` is set to wait in our `queue_enqueue` method when the stack if full. Once the `wait` is called in enqueue, the thread responsible for calling enqueue is set to sleep, the mutex lock is released temporarily and finally, the control is passed to dequeue where at least one element is removed from the queue. Once the element is removed, `dequeue` `signals` the condition variable `ready_to_produce` so that enqueue won't be blocked anymore, and then enqueue can start enqueuing data once again because there is some space in the queue.

Unit-Tests for Queue

Our unit test for Queue can be found in `unit_test/queue_test.c`. We tested the Queue by two main testing approaches. 1. Vanilla Test (no threading) is meant to just enqueue certain number of elements, and dequeue the queue until its empty without the use of any type of threading. So if the Queue is empty and there are no memory leaks then the Vanilla Test ran successfully. 2. Threading Test is designed to test out the producer-consumer model on the Queue. We created a producer worker function that produces/enqueues `n` number of elements in the Queue. We also created a consumer worker function that consumes/dequeues the elements from the Queue until the Queue is empty, and the producer has no more data to produce. Notice that unlike the vanilla test, this test will concurrently enqueue, and dequeue elements from the Queue. This will also test if the pthread synchronization devices work as expected. So if the Queue is empty after every test and there are no memory leaks then the threading Test ran successfully. 3. General conditions for the test. 1. For the threading test, we tried different combinations for the number of producer, and the number of consumer threads that will target the producer worker function, and consumer worker function respectivily. For instance let's say a pair of producer, and consumer threads is represented as `(p,c)`. Then we tested the threading test with the pairs (1,1), (3,5), (5,3), (4,4). 2. For the vanilla, and threading test we tested it with n (10, 100, 1000, 10000) as the bounded queue is of size n. This number can be changed in the main method of `unit_test/pool_test.c`

Utils

The purpose of this component is to share utility functions to ease the development workflow. Our utilility functions can be found in `src/utils.c`.

Util functions

1. `void print_buffer(char *word_buffer, int length);` is used for debugging purposes to print out the buffer in `wrap_text`.
2. `int safe_write(int fd, const void *__buf, long __nbyte)` is a wrapper function built on top of the system call `write`. It checks whether or not `number of bytes` was successfully written into the file. Otherwise, it returns -1 when an error occurs.
3. `int check_file_or_directory(struct stat *file_in_dir_pointer);` returns 1 when if given file is a regular file, returns 2 when if given file is a directory, otherwise it returns 0.
4. `int check_rsyntax(char *r);` checks if user input includes `-r`.
5. `int fill_param_by_user_arguememt(int argv, char **arg, int *max_width, int *producer_threads, int *consumer_threads, int *isrecursive, int *widthindex);` parses user arguements. It parses `-r` option for the number of `producer_threads` and `consumer_threads`. It also parses the provided max_width. However this function won't parse the files or directories.
    - If the user input only consist of `-r` then `producer_threads=1` and `consumer_threads=1`.
    - If the user input only consist of `-rN,` then `producer_threads=1` and `consumer_threads=N`.
    - If the user input only consist of `-rM,N` then `producer_threads=M` and `consumer_threads=N`. Based on if `-r` option was provided, we get the `max_width` arguement from the user input `char** argc` array.
6. `char *concat_string(char *prev_str, char *new_str, int optional_prev_length, int optional_new_length);` it concatenates the first and second strings. If `optional_prev_length` = -1 or `optional_new_length` = -1, the string length for the appropriate string will be computed, else it is the client's responsibility to give a valid string length for the appropriate string. Note the output must be freed by the client at the end.
7. `*append_file_path_to_existing_path` appends the name of a directory or regular file to an existing given directory path. It is also the clients responsibility to free the output of this function.

Unit-Tests for Utils

Our unit test for Utils can be found in `unit_test/utils_test.c`. We only formally tested out `append_file_path_to_existing_path` and `concat_string` as they have a lot of edge cases.

1. `test_append_file_path_to_existing_path`
    - Existing path does not end with connection `/`. i.e. /a/b/c and new path d so path will be /a/b/c/d
    - Existing path does end with connection `/`. i.e. /a/b/c/ and new path d so path will be /a/b/c/d
2. `test_concat_strings`
    - Concat two string of length s1, s2 where s1 < s2.
    - Concat two string of length s1, s2 where s1 > s2.
    - Concat two string of length s1, s2 where s1 = s2.
    - Concat two string of length s1, s2 where s1 = s2 = 0
    - Concat two string of length s1, s2 where s1 = 1 and s2 = 0
    - Concat two string of length s1, s2 where s1 = 0 and s2 = 1

Word Break

This is the program where all of the previous components like the bounded queue, unbounded stack, util functions get combined into a multithreaded word wrapping program. We tried to use the best modularization practices, however, if you have any suggestions please feel free to comment it.

We used a producer-consumer approach to tackle the multithreaded word-wrap program. Our producer worker function `void *produce_files_to_wrap(void *arg);` is targetted by the producer/directory thread and our consumer worker function `void* consume_files_to_wrap(void *arg);` is targetted by the consumer/wrapping threads. Our user interface allows us to control how many producer or how many consumer threads we desire, however you can read more about that in our user interface section.

produce_files_to_wrap

`produce_files_to_wrap` takes in a `struct file_producer` as the producer thread arguement. In that struct we have a `file_queue` field that is the bounded file queue responsible for enqueuing all the files from the directories and sub-directories. Then we have a field `dir_pool` which is our unbounded directory stack that is responsible for recursively enqueuing/dequeuing directories and sub directories. We also have a field `isrecursive` to enable recursive directory traversal or not (for the extra credit). Finally we have a `error_code` field that is an integer pointer and stores any error that occured in the producer worker function.

The directory pool initially is `not` passed empty to the `produce_files_to_wrap` function because we need a initial parent directory or some parent directories (extra credit) that the producer worker function can start traversing (look at next section regarding `handle_multiple_input_files`). In the loop `produce_files_to_wrap` function first `dequeues` a `parent_dir_path` from the `dir_pool` by calling the `pool_dequeue` function. Then a helper function `int fill_pool_and_queue_with_data(char *parent_dir_path, Pool *dir_pool, Queue *file_q, int isrecursive);` is called and it is responsible for taking the recently dequeued `parent_dir_path` and filling up the `dir_pool` and `file_q` with sub-directories and regular files from the `parent_dir_path`. Internally, `fill_pool_and_queue_with_data` calls `pool_enqueue` function to enqueue sub-directories in the `dir_pool` and `queue_enqueue` function to enqueue regular file path in the `file_q`. One more thing to mention, `fill_pool_and_queue_with_data` function also takes a `isrecursive` option which basically if `not` enabled then it indicates `fill_pool_and_queue_with_data` to not insert any sub-directory in the `dir_pool`. This `isrecursive` option was specifically made for the extra credit section because the user interface can disable recursive directory traversal.

int handle_multiple_input_files(int widthindex, int max_width, int argv, char **arg, Pool *dir_pool);

This function is specifically made to parse the files or directories the user provides. Since we are doing the extra credit assignment, we also add support to handle multiple files and directories. This function will loop through all the remaining arguments after the `max_width` arguement. If the argument is a regular file, it will use the `wrap_text()` function. If it is a directory, it will be enqueued in the directory_pool (`*dir_pool`) by the `pool_enqueue` function. When the directory is enqueued into the `*dir_pool` it gives some data for the producer worker function to start working with as described in the previous `produce_files_to_wrap` section. If there is only one regular file provided, it wraps and writes to `stdout` . If at least one of the file has a invalid path then the function returns `EXIT_FAILURE` immediately.

int fill_pool_and_queue_with_data(char *parent_dir_path, Pool *dir_pool, Queue *file_q,int isrecursive);

How fill_pool_and_queue_with_data fills up data in the queue and pool?

To fill the data in the file queue, `fill_pool_and_queue_with_data` allocates a object of type `queue_data_type` on the heap, and this object stores the regular `input_file` and `output_file` . Since the output file needs to be computed from the `input_file` , we create easy to use functions called `concat_string` and `append_file_to_existing_path` in our `util.c` file to ease the creation of the output file. In short, the `concat_string` concats `wrap.` extension to the existing `input` file name and `append_file_to_existing_path` concats the existing directory to the new output filename. This input filename and output filename together is enqueued as a `queue_data_type` object into the file queue for the consumer threads to later dequeue and wrap the input file to the output file. Moreover to fill the data in the directory stack, `fill_pool_and_queue_with_data` allocates a object of type `pool_data_type` on the heap, and this object stores the `directory_path` . The sub-directories in the directory are appended using the `append_file_to_existing_path` method, and stored in a `pool_data_type` object. This object is finally enqueued in the directory pool using the `pool_enqueue` method,.

Ending condition for produce_files_to_wrap

The ending condition for our producer/directory threads are 1) the directory queue must be empty and 2) the number of directory threads that are actively reading a directory must be 0. To check if directory queue is not empty, we called our function `pool_is_empty` function from `pool.c` , which basically checks if the number of elements in the pool is 0. Moreover, to check the number of directory threads, we use the field `number_of_active_producers` in the Pool struct to keep track of the number of directory threads that are currently working with a directory. So if a directory path is successfully dequeued from the `dir_pool` then the directory thread has some directory to work with and therefore the `number_of_active_producers` count is incremented by calling the `increment_active_producers` . However, once `fill_pool_and_queue_with_data` finishes reading the directory, and fills up the stack and queues appropriately then the `number_of_active_producers` is decremented to indicate that the directory thread working on that particular directory has finished its work. So before we dequeue (the condition to start a directory thread working), we just check if the directory queue is empty and the `number_of_active_producers` is 0 then we just exit the loop and close the directory pool. We close the directory pool to just indicate that there is no more `new` directories so whichever directory thread was in the process of reading a directory should just finish their directory traversal and exit when they are done.

consume_files_to_wrap

`consume_files_to_wrap` takes in a `struct file_consumer` as the consumer thread arguement. In that struct we have a `file_queue` field that is the bounded file queue populated with the regular file paths created by the producer threads. Then we have a `max_width` field that is an argument to the `wrap_text` function inside the `consume_files_to_wrap` . Finally we have a `error_code` field that is an integer pointer and stores any error that occured in the consumer worker function.

The main purpose of this cosumer worker function is to `dequeue` the data from the provided `file_queue` by calling `queue_dequeue` function, and to wrap the regular file path by calling the `wrap_text` algorithm written in the previous project. The item dequeued from the `file_queue` is of type `queue_data_type` and it consists of the `input-file-path` , and `output-file-path` . This input and output file paths are passed to the `wrap_text` function and a new wrapped file is written to `output-file-path` . Since the producer loans the consumer an allocated heap-space item of type `queue_data_type` , so it is also the responsiblity of this consumer function to free this object.

General Properties of the wrap_text Algorithm (Taken from our project 2 readme)

For Word Wrap to `Wrap Correctly` the following properties must be satisfied. - The bytes read from an input file must be ordered correctly in the wrapped output file. - The number of characters in each line excluding the newline character of the wrapped output file must be less than or equal to the `max_width` . - Each word in a line must be seperated by a space. Words cannot be broken down to a sequence of characters from one line to another. If the word can't fit in the line then its moved to the next line. - Each line must end with a newline including the last line. - The lines are normalized meaning extra white spaces in the line is replaced by a single white space, and the line is trimmed so that there is no white space in the beginning or end of a sentence. - The breaks between lines are also normalized meaning extra breaks between consecutive lines is replaced with only one break.

Ending condition for consume_files_to_wrap

The ending condition for our consumer/wrapping threads are 1) The file queue must be empty 2) the file queue must also be closed. To check if directory queue is not empty, we called our function `queue_is_empty` function from `queue.c` , which basically checks if the number of elements in the queue is 0. Moreover, since the file queue is closed only when the directory threads have finished traversing through all directories and sub directories, the consumers won't finish "early" and exit due to this condition because we want all the regular files from every given sub-directory and directory to be wrapped.

Error Handeling.

We check for error codes everywhere in our code. In our producer worker function or consumer worker, if an error was bound to occur in at least one of the running threads then the error code is recorded in the `error_code` arguement that is passed initially to the respective worker functions and `EXIT_FAILURE` is returned when all threads have been joined. The error code is passed to the thread join function through `pthread_exit` . Errors such as (unreadable input file, unable to create output file, word longer than line width) were recorded as an example.

Testing Word Break

In order to test word break thoroughly, we made sure to stress tested our program with multiple levels of sub-directories, a single level directory and wrong directory name. As an extra effort, we also wrote a script in Python to generate multiple levels of sub-directories in order to test the program to its extreme too! We combined all three parts of the projects into one part by creating a common user interface. For instance if a user executes `./wordbreak -r dir_test/` , the program would run with 1 producer and 1 consumer thread. If the user executes `./wordbreak -rN dir_test/` , the program would run with 1 producer thread and N consumer threads. Finally if the user executes `./wordbreak -rM,N dir_test/` , then there will M producer threads and N consumer threads. Since we don't have a formal way of testing whether the project is really multithreading, so we used a custom `debug_print` macro that is internally built using `fprintf` and it is located in `logger.h` . We then used this macro to print out the thread ids, when the files or directories were about to get enqueued/dequeued. We also made sure in our debug logs that a regular file was not wrapped more than once in the entire log output file. We also wrote a script in python to check if a all regular files in the directories and sub directories do have a corrosponding wrap file.

Testing the wrap_text algorithm

1. Empty file (`0 bytes`)
    - **Result**: The program return an empty file.
2. Paragraph with a single `break`
    - **Result**: Wraps correctly, and the single break remains as a single break.
3. Paragraph with multiple `breaks`
    - multiple single breaks between sentence
        - **Result**: Wraps correctly, and single break remains a single break.
    - multiple multiple breaks between sentence.
        - **Result**: Wraps correctly, and multiple breaks become a single break
4. One word per line.
    - **Result**: Wraps correctly. Each word in the line ends with a newline character. If the word length exceeds the `max_width` then `EXIT_FAILURE` is returned internally by the process (look at point 4 in Testing the errors section).
5. Skipping files that start with `wrap.` or `.`
    - **Result**: Returns Nothing, but skips over the file when traversing the directory
6. Varying read buffer-size does not affect the algorithm.
    - **Result**: Wraps correctly.
7. Abnormal whitespace locations in random sentence of the paragraphs.
    - **Result**: Normalizes the sentence by removing any extra white spaces in the trailing, leading or any portion of the sentence.
8. Large `max_width` value must wrap all lines into `one` single line.
    - **Result**: Wraps correctly.
9. Rewrapping a file with the same `max_width` used to wrap that file with the same `max_width` before.
    - **Result**: Wraps correctly. Identical file is returned. If the `cmp <wrapped-file> <re-wrapped-file>` program returns nothing then `<wrapped-file>` and `<re-wrapped-file>` is a identical file.
10. Minimum non-empty file (only 1 byte).
    - **Result**: Wraps correctly. The output has two characters, the existing 1 byte and the terminating newline character.
11. A file with only non-alphanumeric characters (e.g. \t, \n, etc).
    - **Result**: Wraps correctly. The output is 0 bytes as there were no alphanumeric characters in the input at all.

Testing the errors

1. Invalid file or directory path
    - **Result**: Invalid File Path Error and returns `EXIT_FAILURE`
2. Reading files with no read permission
    - **Result**: Permission Denied Error Message and returns `EXIT_FAILURE`
3. Writing files with no write permission
    - **Result**: Permission Denied Error Message and returns `EXIT_FAILURE`
4. Words that exceed `max_width`
    - **Result**: Program continues to wrap the file, however the process that's executing `word_break` will return an `EXIT_FAILURE` . This can be checked by executing the program `echo $?` to see whether the latest executed program returned an `EXIT_FAILURE` or `EXIT_SUCCESS` .

Testing Extra Credit

1. Empty file (`0 bytes`)
    - **Result**: The program return an empty file.
2. `./bin/word_break -r note.txt`
    - **Result**: fill_param_by_user_arguememt(): Max width was either not provided or it cannot be 0!
3. `./bin/word_break -r 0 note.txt`
    - **Result**: Error w./bin/word_break -r 0 note.txt fill_param_by_user_arguememt(): Max width was either not provided or it cannot be 0! main(): Error with parsing arguements.
4. `./bin/word_break -r 10 note.txt`
    - **Result**: It wraps the 10 characters per line, creates `wrap.note.txt` file and writes to standart output recursively. It uses 1 `*producer_threads` and 1 `*producer_threads` .
5. `./bin/word_break -r20, 10 note.txt`
    - **Result**: It wraps the 10 characters per line, creates `wrap.note.txt` file and writes to standart output recursively. It uses 1 `*producer_threads` and 20 `*consumer_threads` .
6. `./bin/word_break -r20, 10 note.txt`
    - **Result**: It wraps the 10 characters per line, creates `wrap.note.txt` file and writes to standart output recursively. It uses 1 `*producer_threads` and 20 `*consumer_threads` .
7. `./bin/word_break -r20,5 10 note.txt`
    - **Result**: It wraps the 10 characters per line, creates `wrap.note.txt` file and writes to standart output recursively. It uses 20 `*producer_threads` and 5 `*producer_threads` .
8. `./bin/word_break -r20,5 10 note.txt`
    - **Result**: It wraps the 10 characters per line, creates `wrap.note.txt` file and writes to standart output recursively. It uses 20 `*producer_threads` and 5 `*producer_threads` .
9. `./bin/word_break -r 10 note.txt commands.txt`
    - **Result**: It wraps the 10 characters per line in created `wrap.note.txt` and `wrap.commands.txt` files recursively. It uses 1 `*producer_threads` and 1 `*producer_threads` .

10. `./bin/word_break -r3, 20 note.txt commands.txt`
    - **Result**: It wraps the 20 characters per line in created `wrap.note.txt` and `wrap.commands.txt` files recursively. It uses 1 `*producer_threads` and 3 `*producer_threads`.
11. `./bin/word_break 20 note.txt commands.txt`
    - **Result**: It wraps the 20 characters per line in created `wrap.note.txt` and `wrap.commands.txt` files without using multithreads.
12. `./bin/word_break 20 tests`
    - **Result**: It wraps the 20 characters per line in created all the `wrap.*` version of the regular files in the `tests` directory files without using multithreads, but it does not create `wrap.*` version of the files in the sub directories such as `tests/foo`, `tests/foo/foo_a`, `tests/foo/foo_b`.
13. `./bin/word_break -r3,5 20 tests`
    - **Result**: It wraps the 20 characters per line in created all the `wrap.*` version of the regular files including its subdirectory regular files in the `tests` directory recursively. It uses 3 `*producer_threads` and 5 `*producer_threads`.
14. `./bin/word_break -r3,5 20 tests/foo tests2`
    - **Result**: It wraps the 20 characters per line in created all the `wrap.*` version of the regular files including its subdirectory regular files in the `tests/foo` directory recursively.It also wraps the 20 characters per line in created all the `wrap.*` version of the regular files including its subdirectory regular files in the `tests2` directory recursively. It uses 3 `*producer_threads` and 5 `*producer_threads`.
15. `./bin/word_break -r3,5 20 tests/foo/foo_a/foo_c/a.txt tests2`
    - **Result**: It wraps the 20 characters per line in created `tests/foo/foo_a/foo_c/a.txt` file recursively. It also wraps the 20 characters per line in created all the `wrap.*` version of the regular files including its subdirectory regular files in the `tests2` directory recursively. It uses 3 `*producer_threads` and 5 `*producer_threads`.
16. `./bin/word_break -r3,5 20 tests tests2 tests3`
    - **Result**: It wraps the 20 characters per line in created all the `wrap.*` version of the regular files including its subdirectory regular files in the `tests` `tests2` `tests3` directory recursively. It also wraps the 20 characters per line in created all the `wrap.*` version of the regular files including its subdirectory regular files in the `tests` `tests2` `tests3` directory recursively. It uses 3 `*producer_threads` and 5 `*producer_threads`.
17. `./bin/word_break 20 tests tests2 tests3`
    - **Result**: It wraps the 20 characters per line in created all the `wrap.*` version of the regular files including its subdirectory regular files in the `tests` `tests2` `tests3` directories without using multithreads. It does not create `wrap.*` version of the files in the sub directories for `tests` `tests2` `tests3` directories.

Steps To Run Word Break.

1. First make to sure to turn on or off the `DEBUG` parameter located in `src/logger.h`. `DEBUG` 0 will not print any logs and 1 will print all the logs to stdout.
2. Then change the initial `POOLSIZE` or `QUEUESIZE` macro in `src/word_break.h` to setup the initial poolsize or queuesize.
3. Then run `make` from the project root directory to generate the object files and binaries. Note the object files were made seperately in order to unit test them independently of each other.
4. A binary for `word_break` is created in the bin folder. Now if a user executes `./word_break -r dir_test/`, the program would run with 1 producer and 1 consumer thread. If the user executes `./word_break -rN dir_test/`, the program would run with 1 producer thread and N consumer threads. Finally if the user executes `./word_break -rM,N dir_test/`, then there will M producer threads and N consumer threads.
5. You can also not include the `-r` option but then the program will not recursively traverse the input directory.
6. If you want to run the extra credit part, it is the same instructions as number 4 except you can include multiple directories or files.
7. `make fresh file=test_dir/` to clear all wrap files out of a directory
8. `make clean` to clean build.

Steps to Run Unit Test

1. First make to sure to turn on or off the `DEBUG` parameter located in `src/logger.h`. `DEBUG` 0 will not print any logs and 1 will print all the logs to stdout.
2. Then change the initial `POOLSIZE` or `QUEUESIZE` macro in `src/word_break.h` to setup the initial poolsize or queuesize.
3. Then run `make` from the project root directory to generate the object files and binaries. Note the object files were made seperately in order to unit test them independently of each other.
4. Now cd into `unit_tests` and run `make` from there to generate the unit test binaries.
5. Then you can run `./bin/pool_test` or `./bin/queue_test` or `./bin/utils_test` to test out the components invidually.

Terminology

1. Producer threads are the same as directory threads.
2. Consumer threads are the same as wrapping threads.
3. a `break` is defined as two newlines between a consecutive sentence

Authors

Rohan Deshpande (ryd4) and Selin Denise Altiparmak (sda81)