CSE 344 - HW5 Report

ERSEL CELAL EREN 1901042673

In this homework assignment, we are asked to create a multithreaded file copying program. As an overview, I get buffer size, number of thread, source directory and destination directory from user as command-line arguments.

>>> Main function <<<

Buffer and Memory Allocation:

The code dynamically allocates memory for the buffer array, which will store the tasks (file copying information) for the worker threads.

Starting Time Measurement:

The code uses the gettimeofday function to measure the starting time before the traversal and copying process begins.

Destination Directory Creation:

The program tries to create the destination directory using the mkdir function.

If the directory already exists, it checks if the last folder name from the source directory can be appended to the destination directory to create a new subdirectory within it.

Thread Initialization:

The program creates an array of worker threads (threads) based on the num_threads argument.

It initializes mutexes (buffer_mutex and output_mutex) and condition variables (buffer_not_full and buffer_not_empty) used for synchronization.

Root Thread Creation:

The program creates a root thread (root_thread) that is responsible for traversing the source directory and initiating the copying process.

The root thread is created using the pthread_create function and the traverse_directory function is passed as the thread routine.

The args array, containing the source directory and destination directory, is passed as an argument to the root thread.

Worker Thread Creation:

The program creates multiple worker threads (specified by num_threads) using a loop.

Each worker thread is created using the pthread_create function, and the worker_thread function is passed as the thread routine.

Thread Joining and Cleanup:

The program waits for the root thread to finish its execution using pthread_join.

>>> traverse_directory(...) <<<

Directory Opening:

This function attempts to open the source directory using the ‘opendir’ function.

If the directory cannot be opened, an error message is printed, and the function returns.

Directory Traversal:

It enters a loop that iterates over the entries in the source directory using the ‘readdir’ function.

For each entry, the function checks if it represents a regular file, a directory (excluding "." and ".."), or a FIFO (named pipe) file.

To determine the type of the entry, the stat or lstat function is used.

If the entry is a regular file or a FIFO file, it is considered a file that needs to be copied to the destination directory.

A file copying task is created for the entry, which typically includes the source path, destination path, and any additional information required for the copying process.

The file copying task is enqueued into the shared buffer or task queue, which will be processed by the worker threads responsible for copying the files.

If the entry is a directory, it needs to be traversed recursively.

A recursive call to traverse_directory is made with the new source and destination paths constructed based on the current entry.

The function will effectively enter the subdirectory and continue the traversal from there.

It starts with the source directory, processes each entry encountered, and if it's a directory, it explores it by making a recursive call. This process continues until all files and directories within the source directory have been visited. The function enqueues file copying tasks for files and FIFOs, ensuring that the worker threads will handle the copying process.

>>> add_task_to_buffer(...) <<<

Mutex Lock:

The function begins by acquiring the lock on the buffer_mutex using the pthread_mutex_lock function.

This ensures that only one thread can access the buffer at a time, preventing race conditions or conflicts when modifying the buffer.

Buffer Full Check:

The function enters a while loop that checks if the buffer is full.

The condition ((buffer_in + 1) % buffer_size) == buffer_out checks if the next position in the buffer (after adding the task) would be the same as the current position of buffer_out.

If the buffer is full, meaning the next position would overwrite an unprocessed task, the thread waits for the buffer_not_full condition variable to be signaled.

The pthread_cond_wait function releases the lock on buffer_mutex and puts the thread to sleep until the condition variable is signaled. When the thread wakes up, it reacquires the lock before continuing.

Task Addition:

Once the buffer has space for a new task, the function proceeds to add the task to the buffer.

The function assigns the source file descriptor src_fd, destination file descriptor dest_fd, and filename filename to the current buffer_in position.

The buffer_in index is then incremented by one and wrapped around to the beginning of the buffer using (buffer_in + 1) % buffer_size.

Signal and Unlock:

After adding the task to the buffer, the function signals the buffer_not_empty condition variable using pthread_cond_signal.

This signal notifies any waiting threads that the buffer is no longer empty, and there are tasks available for processing.

Finally, the lock on buffer_mutex is released using pthread_mutex_unlock, allowing other threads to access the buffer.

This function is used to add a new task, represented by the source file descriptor, destination file descriptor, and filename, to a shared buffer. It ensures that the buffer is not full before adding the task and signals the availability of new tasks to any waiting threads. The use of mutex and condition variables helps synchronize access to the shared buffer and coordinate the producer-consumer relationship between the task-adding thread and the worker threads.

>>> worker_thread(…) <<<

The worker_thread function is designed to be executed concurrently by multiple worker threads. Each worker thread continuously checks for tasks in the shared buffer, retrieves a task, performs the task, and repeats the process. The use of mutex and condition variables ensures synchronized access to the shared buffer, allowing the worker threads to coordinate with the task-adding thread and process tasks efficiently without conflicts.

This loop allows the worker thread to continuously process tasks until the program is terminated.

The function begins by acquiring the lock on the buffer_mutex using the pthread_mutex_lock function.

This ensures that only one thread can access the buffer at a time, preventing race conditions or conflicts when modifying the buffer.

Buffer Empty Check:

The function enters a while loop that checks if the buffer is empty.

The condition buffer_in == buffer_out checks if the buffer has no pending tasks to process.

If the buffer is empty, meaning no tasks are available for processing, the thread waits for the buffer_not_empty condition variable to be signaled.

The pthread_cond_wait function releases the lock on buffer_mutex and puts the thread to sleep until the condition variable is signaled. When the thread wakes up, it reacquires the lock before continuing.

Task Retrieval:

Once there is a task available in the buffer, the function retrieves the task from the buffer_out position.

It assigns the source file descriptor src_fd, destination file descriptor dest_fd, and filename filename from the buffer to local variables.

The buffer_out index is then incremented by one and wrapped around to the beginning of the buffer using (buffer_out + 1) % buffer_size.

Signal and Unlock:

After retrieving the task from the buffer, the function signals the buffer_not_full condition variable using pthread_cond_signal.

This signal notifies any waiting threads that the buffer is no longer full and there is space available for new tasks.

Finally, the lock on buffer_mutex is released using pthread_mutex_unlock, allowing other threads to access the buffer.

Perform Task:

Once the task is retrieved and the lock is released, the worker thread proceeds to perform the task.

In this case, the task is represented by the copy_file function, which copies a file from the source file descriptor src_fd to the destination file descriptor dest_fd, using the provided filename filename.

Loop Continuation:

After completing the task, the worker thread returns to the beginning of the loop, reacquiring the lock on buffer_mutex and checking for new tasks in the buffer.

If there are no tasks available, the thread will wait for the buffer_not_empty condition variable to be signaled, and the process repeats.

>>> copy_file(...) <<<

The copy_file function reads data from the source file descriptor in chunks and writes it to the destination file descriptor until there is no more data to read. It handles errors that may occur during the read and write operations and provides feedback on the success or failure of the file copying process.

Buffer Initialization:

The function declares a character array buffer of size BUFSIZE, which will be used to read and write data in chunks.

Read and Write Loop:

The function enters a loop that continues until the read function returns a value of 0, indicating that there is no more data to be read from the source file descriptor (src_fd).

Inside the loop, it reads data from the source file descriptor using the read function, which reads up to BUFSIZE bytes from the file descriptor into the buffer.

The number of bytes read is stored in the bytes_read variable.

Write to Destination File Descriptor:

After reading data from the source file descriptor, the function writes the data to the destination file descriptor (dest_fd) using the write function.

The write function writes the data in the buffer to the file descriptor, using the number of bytes read (bytes_read) as the length.

Error Handling:

If the number of bytes written (bytes_written) is not equal to the number of bytes read (bytes_read), it indicates an error occurred while writing to the destination file.

In this case, the function prints an error message using perror and returns, performing any necessary error handling and cleanup.

Check for Read Error:

After the read-write loop, the function checks if the read function encountered an error by checking if bytes_read is less than 0.

If bytes_read is less than 0, it indicates an error occurred while reading from the source file.

In this case, the function prints an error message using perror and returns, performing any necessary error handling and cleanup.

File Descriptor Closure:

If the copy operation is successful, the function closes both the source and destination file descriptors using the close function.

Closing file descriptors is important to release system resources and prevent resource leaks.

>>> process_task_from_buffer(...) <<<

This function is responsible for retrieving a task from the buffer, updating the buffer indices, and processing the task by invoking the copy_file function. It ensures synchronization using mutex locks and condition variables to handle cases where the buffer is empty or full, and allows multiple worker threads to cooperatively process tasks from the shared buffer.

Buffer Access and Synchronization:

The function begins by acquiring the buffer_mutex lock using pthread_mutex_lock to ensure exclusive access to the shared buffer and associated variables.

Wait for Task in Buffer:

The function enters a loop and checks if the buffer is empty. If the buffer is empty (i.e., buffer_in is equal to buffer_out), it waits on the buffer_not_empty condition variable using pthread_cond_wait.

The pthread_cond_wait function releases the buffer_mutex and suspends the execution of the thread until it is signaled by another thread that a task has been added to the buffer.

Retrieve Task Details:

Once a task is available in the buffer, the function retrieves the source file descriptor (src_fd), destination file descriptor (dest_fd), and filename associated with the task.

The values are obtained from the buffer at the index buffer_out.

Update Buffer Index:

After retrieving the task details, the function updates the buffer_out index by incrementing it using modulo arithmetic to wrap around the buffer.

This indicates that the task has been processed and the corresponding buffer slot can be reused.

Signal Buffer Availability:

Before releasing the buffer_mutex, the function signals the buffer_not_full condition variable using pthread_cond_signal.

Signaling buffer_not_full notifies other threads waiting to add tasks to the buffer that there is now space available.

Release Buffer Lock:

The function releases the buffer_mutex using pthread_mutex_unlock to allow other threads to access the buffer.

Task Processing:

After releasing the buffer lock, the function proceeds to process the retrieved task.

It first prints a message indicating the worker ID (worker_id) and the filename being copied.

Call copy_file Function:

The function calls the copy_file function, passing the source file descriptor (src_fd), destination file descriptor (dest_fd), and filename as arguments.

The copy_file function performs the actual file copying operation.

>>> copy_fd(…) <<<

The copy_fd function performs the actual copying of data from the source file descriptor to the destination file descriptor. It uses a buffer to read and write data in chunks, ensuring that the data is properly transferred. The function handles errors that may occur during the read and write operations, providing feedback to the user and allowing for appropriate error handling and cleanup.

Buffer and Variables:

The function declares a character array buffer of size BUFSIZE to store the data read from the source file descriptor and write to the destination file descriptor.

It also declares variables bytes_read and bytes_written of type ssize_t to keep track of the number of bytes read and written in each iteration.

Copy Loop:

The function enters a loop to continuously read from the source file descriptor and write to the destination file descriptor until the end of the file is reached or an error occurs.

In each iteration, it reads data from the source file descriptor using the read system call, which returns the number of bytes read.

The data is read into the buffer with a maximum size of BUFSIZE.

Write to Destination:

After reading from the source file descriptor, the function writes the data from the buffer to the destination file descriptor using the write system call.

The bytes_read value indicates the number of bytes to write from the buffer.

The write system call returns the number of bytes actually written.

Check Write Error:

Inside the loop, the function checks if the number of bytes written (bytes_written) is not equal to the number of bytes read (bytes_read).

If they are not equal, it means an error occurred while writing to the destination file descriptor.

In such a case, the function prints an error message to the standard error stream using fprintf, indicating the failure to write to the destination file descriptor.

It may also perform error handling steps, such as closing file descriptors and cleaning up resources, depending on the specific requirements of the application.

Check Read Error:

After exiting the loop, the function checks if the bytes_read value is less than zero.

If so, it means an error occurred while reading from the source file descriptor.

In this case, the function prints an error message to the standard error stream using fprintf, indicating the failure to read from the source file descriptor.

Similar to the write error case, it may perform error handling steps as necessary.

>>> HASHTABLE <<<

To Keep statistics about the number and types of files copied and track of the total number of bytes copied I implemented Hashtable.

>>> ChatGPT <<<

I used ChatGPT for the following parts of homework:
- I get implementation of HashTable structure to keep statistics.

- I used gettimeofday implementation to count the time.

- I used ChatGPT to make my report explanation more clarified

- I modified some small code blocks to make it more clean and easy to read.

- Instead of implementing Queue structure for buffer, I find alternative solutions with circular array.
(buffer_in = (buffer_in + 1) % buffer_size;)

Test Cases

Source directory : SRC | Destination Directory : DEST

./program <buffer_size> <num_thread> /path/SRC /path/DEST

As an expected behavior program, content of SRC will be copied into DEST folder.

Here the test cases and expected behavior according to "cp -R" command

1- DEST folder is already exist and it is empty.

>> SRC is created in DEST and content of org SRC copied to new SRC

2- DEST folder is already exist, it has files in it.

>>Same as 1st case. Program copied SRC into DEST without changing existing files.

3- DEST folder doesn't exist.

>> DEST folder is created in /path and content of SRC copied into DEST.

4- DEST folder exist. In DEST folder there is SRC. SRC has its own files.

>> Files of SRC is updated.

5- DEST folder already exist, in DEST folder there is SRC. SRC is empty.

>> Content of original SRC is copied to existing empty SRC folder.

6- DEST folder already exist, in DEST folder there is SRC. SRC has different files from its original.

>> It preserved existing files and copied original files of SRC files into SRC folder in DEST folder.

7-DEST folder already exist. In DEST folder there is SRC. In SRC folder there is SRC.

>> It preserved inner SRC and copied original SRC files into outer SRC folder.

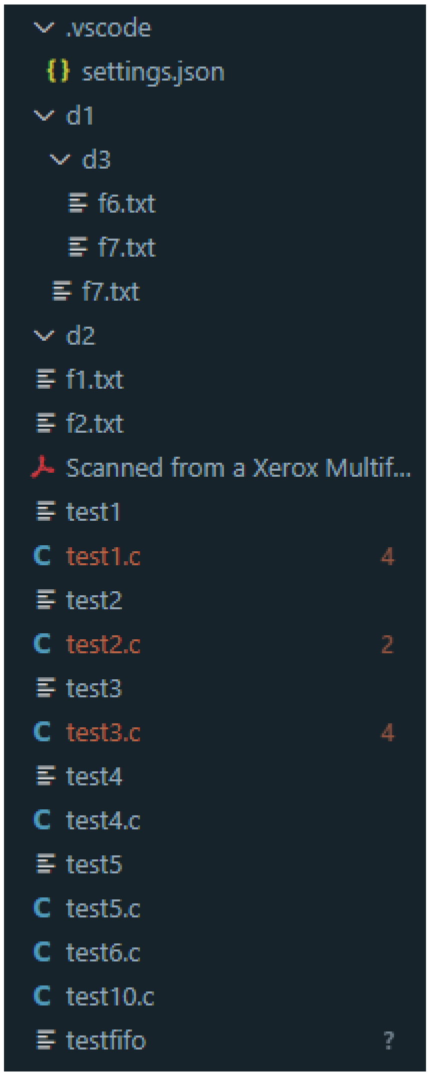
>> I tested this cases with both "cp -R" and my program and I observed that behavior of my program is same as "cp -R" command.

Expected run command :

./program <buffer_size> <num_threads> <source_directory> <destination_directory>

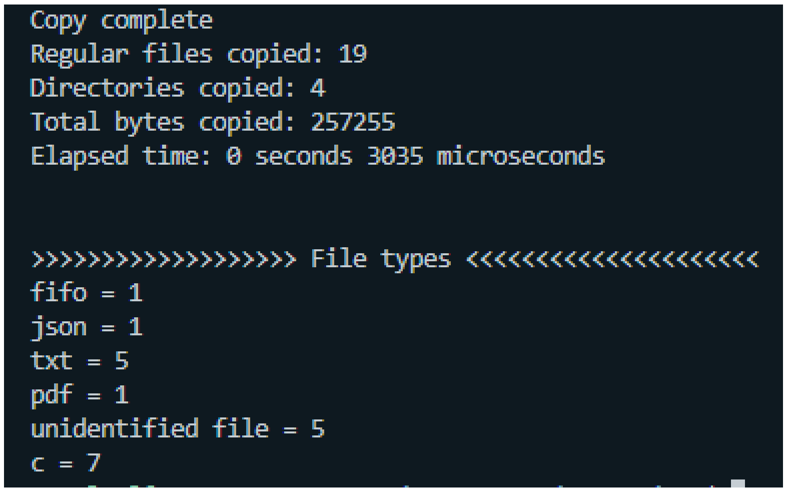
To get path of directories, I used "pwd" command.

Content of SRC folder



```
.vscode
  settings.json
d1
  d3
    f6.txt
    f7.txt
  f7.txt
d2
f1.txt
f2.txt
Scanned from a Xerox Multif...
test1
test1.c        4
test2
test2.c        2
test3
test3.c        4
test4
test4.c
test5
test5.c
test6.c
test10.c
testfifo       ?
```

Output after copy.



```
Copy complete
Regular files copied: 19
Directories copied: 4
Total bytes copied: 257255
Elapsed time: 0 seconds 3035 microseconds


>>>>>>>>>>>>>>>>>>>> File types <<<<<<<<<<<<<<<<<<<<<<
fifo = 1
json = 1
txt = 5
pdf = 1
unidentified file = 5
c = 7
```

There is 19 regular file(.c, .txt, .json, .pdf etc) and 1 FIFO file. I don’t assume FIFOs as regular files. Because of that regular file count is 19. There are 4 folders in SRC directory : .vscode, d1,d2, d3.