ÇAĞRI YILDIZ 1901042630

CSE 344
HOMEWORK 5 REPORT

Summary of Changes from HW4 to HW5

In HW5, the directory copying utility "MWCp" from HW4 has been enhanced with the following features:

1. Condition Variables:

- Used to signal when the buffer is not empty and when the buffer is not full (so the manager can add more items).
- Implemented to ensure that the manager waits when the buffer is full and workers wait when the buffer is empty.

```
34      pthread_cond_t cond_full; // Condition variable to signal buffer is not full
35      pthread_cond_t cond_empty;// Condition variable to signal buffer is not empty
```

2. Barriers:

- Used to ensure that all worker threads wait at a certain point before proceeding. This ensures synchronization of all threads before moving to the next phase of processing.

```
36      pthread_barrier_t barrier; // Barrier for synchronization
```

Project Structure

This report consists of two parts:

- **Project Requirements:** Detailed overview of the assignment's expectations and objectives.
- **Terminal Outputs:** Explanation and demonstration of how the program operates, including descriptions of the outputs observed during its execution.

1. Project Requirements

Objective

The objective of this assignment is to develop a directory copying utility called "MWCp" that copies files and sub-directories in parallel using a worker-manager approach. The program should utilize POSIX and Standard C libraries to manage synchronization and thread activity.

Compilation

The provided code should compile successfully without any errors. To compile the code, you can use the following commands:

- Compile the code: **make**
- Run the code: **make run**
- Check for memory leaks using Valgrind: **make valgrind**

Main Program

- Accepts buffer size, number of workers, and source/destination directories as command-line arguments.

```
98      if (argc != 5) {
99          printf("Usage: %s <buffer size> <number of workers> <source dir> <destination dir>\n", argv[0]);
100         exit(EXIT_FAILURE);
101     }
102
103     // Parse command-line arguments
104     int buffer_size = atoi(argv[1]);
105     num_workers = atoi(argv[2]);
106     char *src_dir = argv[3];
107     char *dest_dir = argv[4];
```

- Starts worker threads and waits for their completion.

```
134     // Create worker threads
135     worker_threads = malloc(num_workers * sizeof(pthread_t));
136     for (int i = 0; i < num_workers; i++) {
137         pthread_create(&worker_threads[i], NULL, worker, (void *)&buffer);
138     }
```

```
152     // Wait for all worker threads to finish
153     for (int i = 0; i < num_workers; i++) {
154         pthread_join(worker_threads[i], NULL);
155     }
```

- Measures execution time to copy files in the directory.

```
120     // Start measuring time
121     struct timespec start, end;
122     clock_gettime(CLOCK_MONOTONIC, &start);
```

```
157     // Stop measuring time
158     clock_gettime(CLOCK_MONOTONIC, &end);
```

- Keeps statistics about the number and types of files copied.

```
295 void print_statistics(int num_workers, int buffer_size, struct timespec start, struct timespec end) {
296     long seconds = end.tv_sec - start.tv_sec;
297     long nanoseconds = end.tv_nsec - start.tv_nsec;
298     long milliseconds = (seconds * 1000) + (nanoseconds / 1000000);
299     long minutes = seconds / 60;
300     seconds = seconds % 60;
301
302     printf("\n--------------STATISTICS-------------------\n");
303     printf("Consumers: %d - Buffer Size: %d\n", num_workers, buffer_size);
304     printf("Number of Regular Files: %d\n", regular_files);
305     printf("Number of FIFO Files: %d\n", fifo_files);
306     printf("Number of Directories: %d\n", directories);
307     printf("TOTAL BYTES COPIED: %ld\n", total_bytes_copied);
308     printf("TOTAL TIME: %02ld:%02ld.%03ld (min:sec.mili)\n", minutes, seconds, milliseconds);
309 }
```

Manager

- Single manager thread.

```
130     // Create the manager thread
131     pthread_t manager_thread;
132     pthread_create(&manager_thread, NULL, manager, (void *)&args);
133
```

- Reads source and destination directory paths.

```
124     // Set up arguments for the manager thread
125     manager_args args;
126     strncpy(args.src_dir, src_dir, sizeof(args.src_dir) - 1);
127     strncpy(args.dest_dir, dest_dir, sizeof(args.dest_dir) - 1);
128     args.buffer = &buffer;
```

- Opens files for reading and creates corresponding files in the destination directory.

```
206             // Handle regular files
207             int src_fd = open(src_path, O_RDONLY);
208             if (src_fd < 0) {
209                 perror("open src");
210                 continue;
211             }
212             int dest_fd = open(dest_path, O_WRONLY | O_CREAT | O_TRUNC, 0644);
213             if (dest_fd < 0) {
214                 perror("open dest");
215                 close(src_fd);
216                 continue;
217             }
```

- If a file already exists in the destination directory with the same name, the file should be opened and truncated.

```
212             int dest_fd = open(dest_path, O_WRONLY | O_CREAT | O_TRUNC, 0644);
```

- Handles errors in opening files by closing both file descriptors and sending an informative message to standard output.

```
208             if (src_fd < 0) {
209                 perror("open src");
210                 continue;
211             }
212             int dest_fd = open(dest_path, O_WRONLY | O_CREAT | O_TRUNC, 0644);
213             if (dest_fd < 0) {
214                 perror("open dest");
215                 close(src_fd);
216                 continue;
217             }
```

- Notifies program completion by setting a done flag and exits.

```
172     // Signal worker threads that the manager is done producing
173     pthread_mutex_lock(&args->buffer->mutex);
174     args->buffer->done = 1;
175     pthread_cond_broadcast(&args->buffer->cond_empty);
176     pthread_mutex_unlock(&args->buffer->mutex);
```

- Manages the buffer (is it empty or full).

```
219             // Lock the buffer and add file information
220             pthread_mutex_lock(&buffer->mutex);
221             while (buffer->count == buffer->buffer_size) {
222                 pthread_cond_wait(&buffer->cond_full, &buffer->mutex);
223             }
224 
225             file_info info = {src_fd, dest_fd, "", ""};
226             strncpy(info.src_name, src_path, sizeof(info.src_name) - 1);
227             strncpy(info.dest_name, dest_path, sizeof(info.dest_name) - 1);
228             buffer->buffer[buffer->in] = info;
229             buffer->in = (buffer->in + 1) % buffer->buffer_size;
230             buffer->count++;
231 
232             // Signal that the buffer is not empty
233             pthread_cond_signal(&buffer->cond_empty);
234             pthread_mutex_unlock(&buffer->mutex);
```

Worker

- Reads file information from the buffer.

```
252         // Lock the buffer and wait if it's empty
253         pthread_mutex_lock(&buffer->mutex);
254         while (buffer->count == 0 && !buffer->done) {
255             pthread_cond_wait(&buffer->cond_empty, &buffer->mutex);
256         }
257         if (buffer->count == 0 && buffer->done) {
258             pthread_mutex_unlock(&buffer->mutex);
259             break;
260         }
261 
262         // Get file information from the buffer
263         file_info info = buffer->buffer[buffer->out];
264         buffer->out = (buffer->out + 1) % buffer->buffer_size;
265         buffer->count--;
266 
267         // Signal that the buffer is not full
268         pthread_cond_signal(&buffer->cond_full);
269         pthread_mutex_unlock(&buffer->mutex);
```

- Copies files from source to destination.

```
277 void copy_file(file_info *info) {
278     char buffer[BUFFER_SIZE];
279     ssize_t bytes_read, bytes_written;
280     while ((bytes_read = read(info->src_fd, buffer, BUFFER_SIZE)) > 0) {
281         bytes_written = write(info->dest_fd, buffer, bytes_read);
282         if (bytes_written != bytes_read) {
283             perror("write");
284             break;
285         }
286         total_bytes_copied += bytes_written;
287     }
288 
289     // Close file descriptors
290     close(info->src_fd);
291     close(info->dest_fd);
292     regular_files++;
293 }
```

- Writes completion status to standard output.

```c
if (bytes_written != bytes_read) {
    perror("write");  // Completion status is handled here
    break;
```

- Handles critical section for writing to standard output.

```c
// Lock the buffer and add file information
pthread_mutex_lock(&buffer->mutex);
while (buffer->count == buffer->buffer_size) {
    pthread_cond_wait(&buffer->cond_full, &buffer->mutex);
}

file_info info = {src_fd, dest_fd, "", ""};
strncpy(info.src_name, src_path, sizeof(info.src_name) - 1);
strncpy(info.dest_name, dest_path, sizeof(info.dest_name) - 1);
buffer->buffer[buffer->in] = info;
buffer->in = (buffer->in + 1) % buffer->buffer_size;
buffer->count++;

// Signal that the buffer is not empty
pthread_cond_signal(&buffer->cond_empty);
pthread_mutex_unlock(&buffer->mutex);
```

- Terminates when signaled.

```c
if (buffer->count == 0 && buffer->done) {
    pthread_mutex_unlock(&buffer->mutex);
    break;
}
```

- Manages the worker thread pool.

```c
// Create worker threads
worker_threads = malloc(num_workers * sizeof(pthread_t));
for (int i = 0; i < num_workers; i++) {
    pthread_create(&worker_threads[i], NULL, worker, (void *)&buffer);
}
```

Error Handling

- Usage Information:
  - Print usage information and exit if command-line arguments are missing or invalid.

```c
// Check for correct number of command-line arguments
if (argc != 5) {
    printf("Usage: %s <buffer size> <number of workers> <source dir> <destination dir>\n", argv[0]);
    exit(EXIT_FAILURE);
}
```

- Signal Handling:
  - Properly handle SIGINT (Ctrl+C) to allow graceful termination.

```c
// Set up signal handler for SIGINT
signal(SIGINT, handle_signal);
```

- Memory Management:
  - Check for memory leaks using valgrind and ensure proper cleanup of resources.

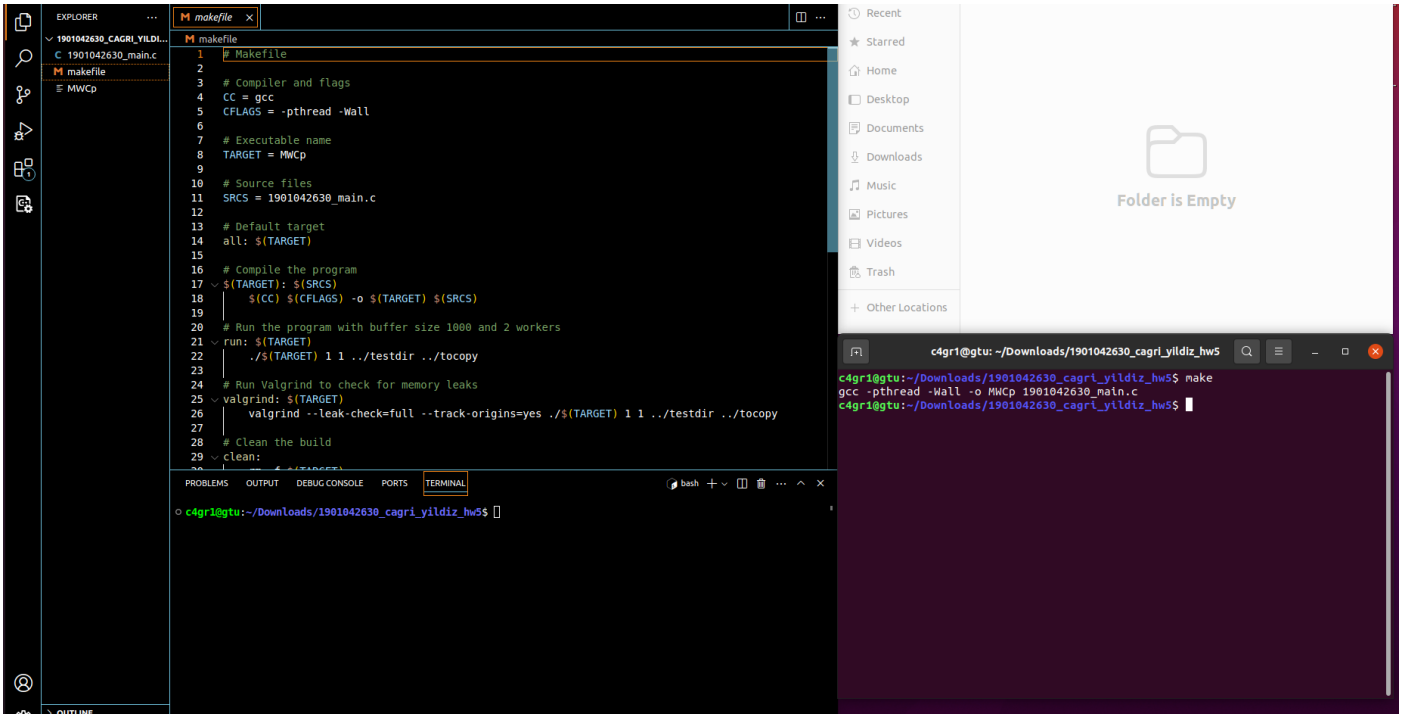
```c
// Function to clean up resources
void clean_up() {
    if (buffer.buffer) {
        free(buffer.buffer);  // Free the buffer memory
    }
    if (worker_threads) {
        free(worker_threads); // Free the worker threads array
    }
    pthread_mutex_destroy(&buffer.mutex);       // Destroy the mutex
    pthread_cond_destroy(&buffer.cond_full);    // Destroy the full condition variable
    pthread_cond_destroy(&buffer.cond_empty);   // Destroy the empty condition variable
}
```

Conclusion

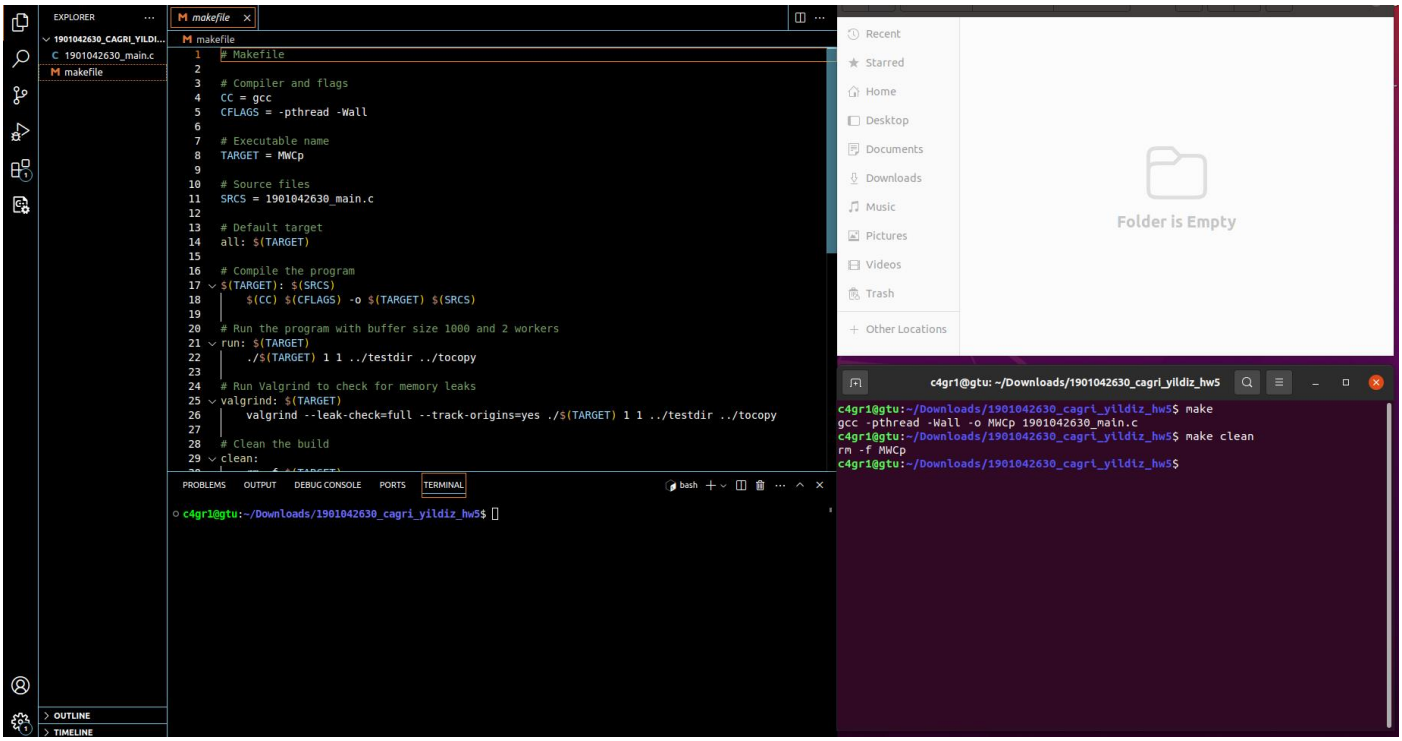
The program meets all the specified requirements and successfully implements a parallel directory copying utility using a worker-manager approach. The detailed code snippets provided demonstrate how the various requirements have been addressed in the implementation.

2. **Terminal Outputs**

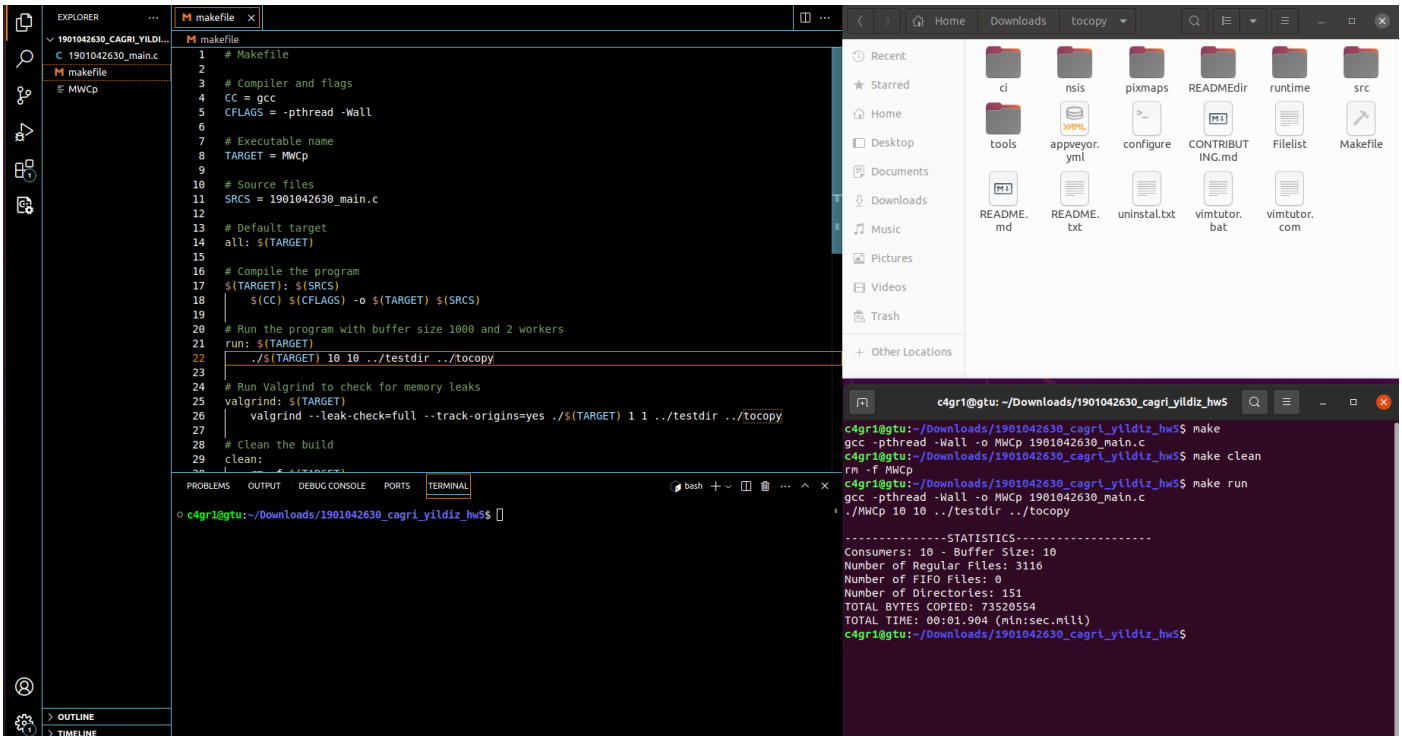
- make or make all



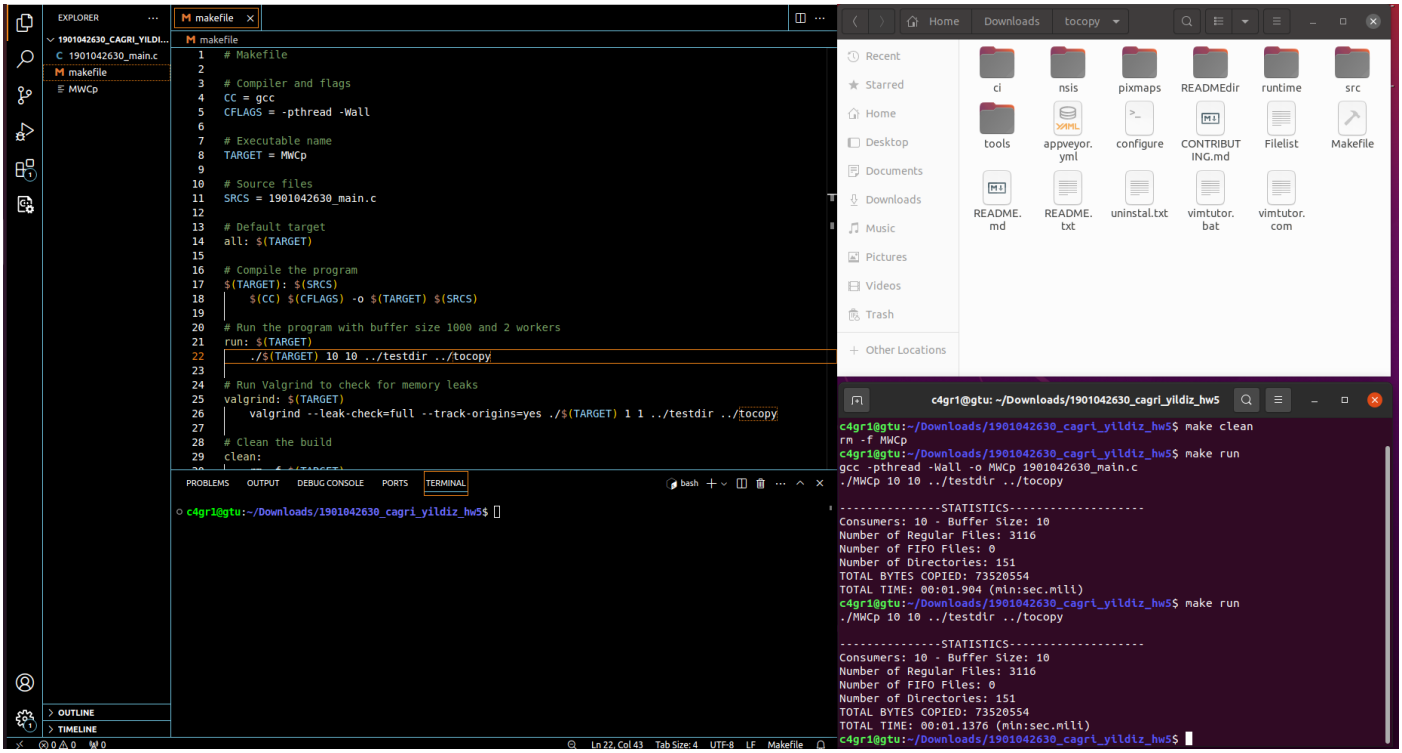
- make clean



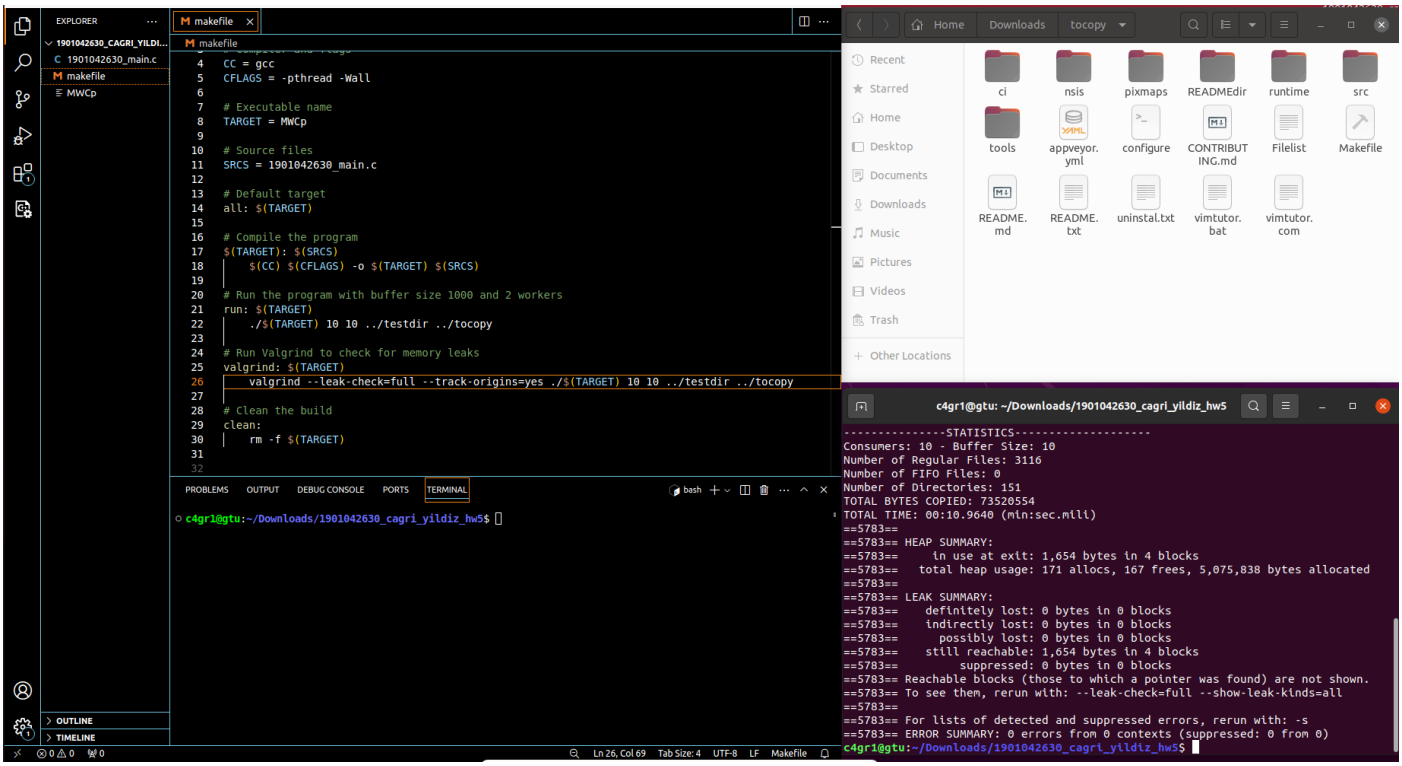
- make run (with 10 10 ../testdir ../tocopy)



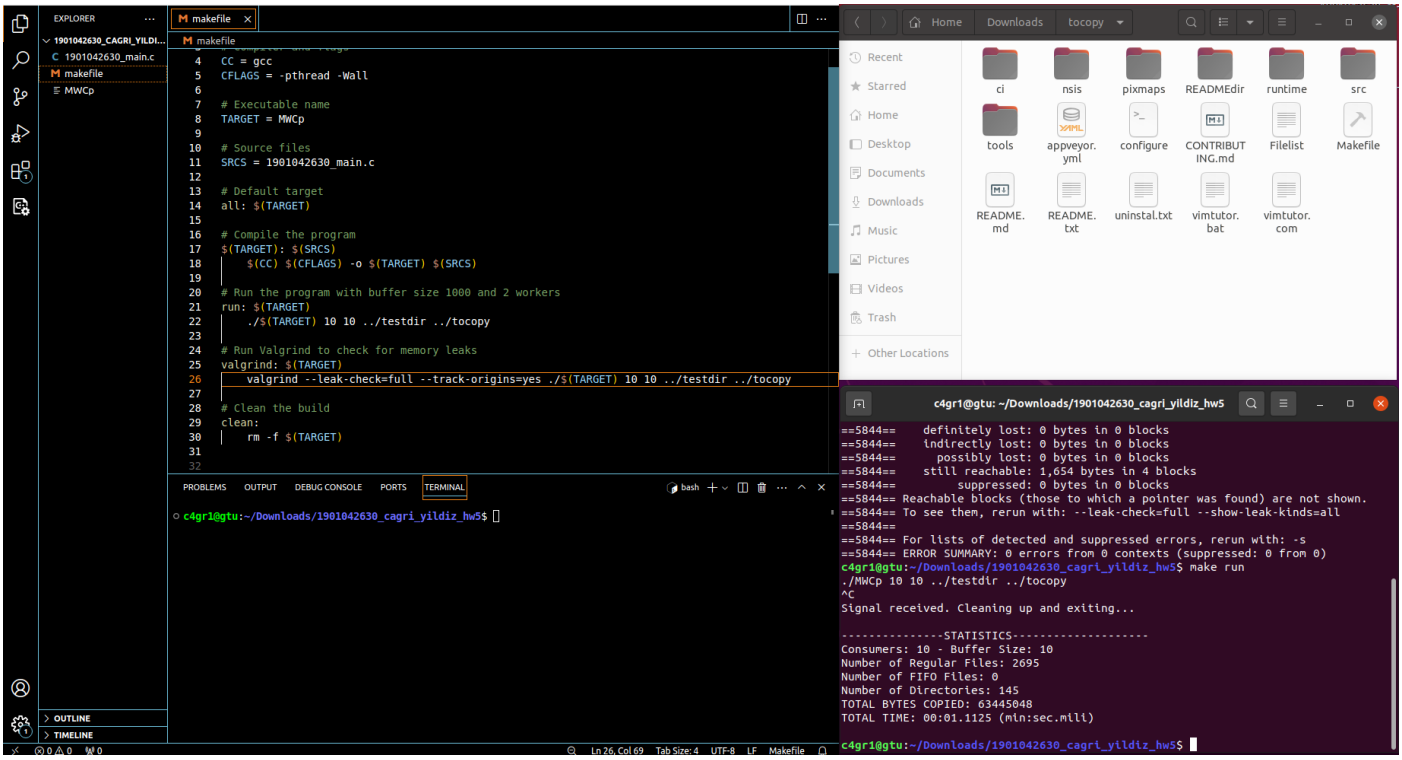
- make run (with 10 10 ../testdir ../tocopy) (re-copying the copied file)



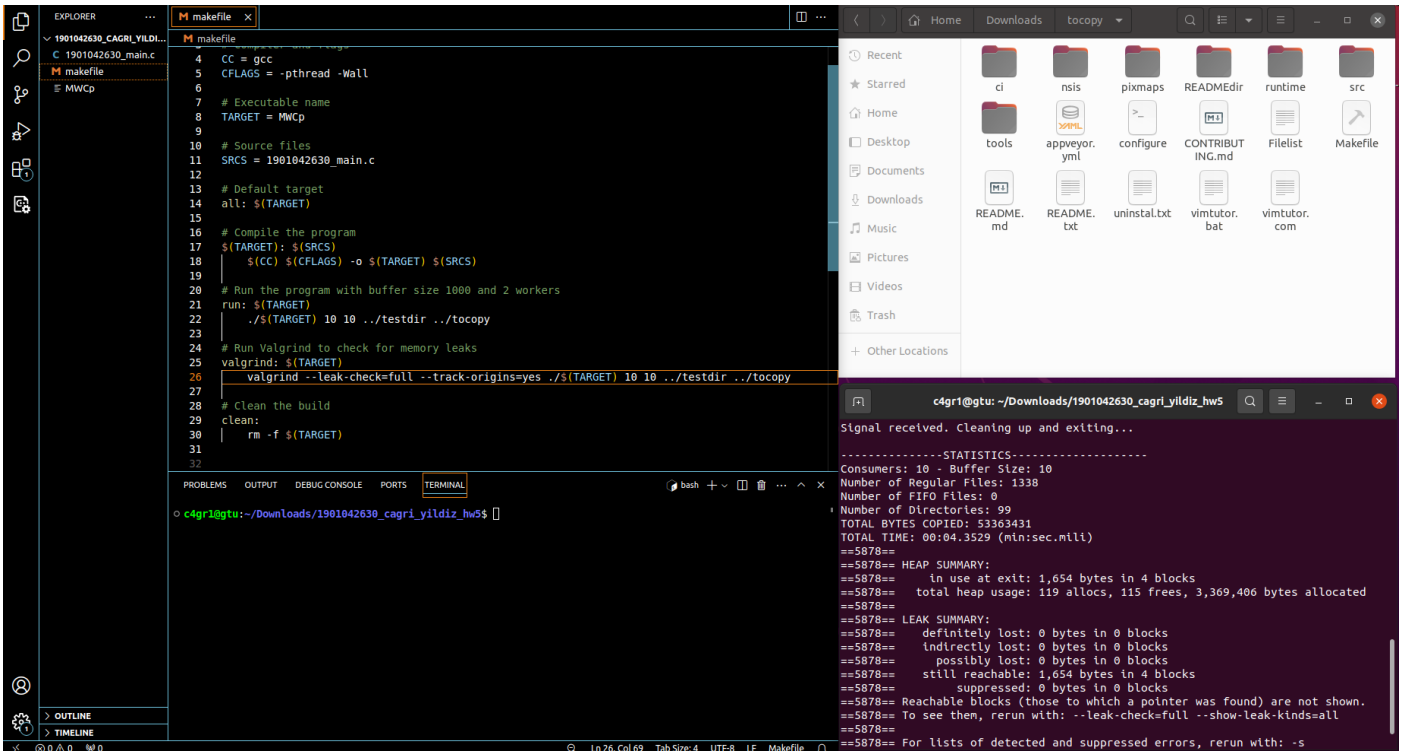
- make valgrind (with 10 10 ../testdir ../tocopy) (re-copying the copied file)



- Make run (with 10 10 ../testdir ../tocopy) (CTRL + C Handling)

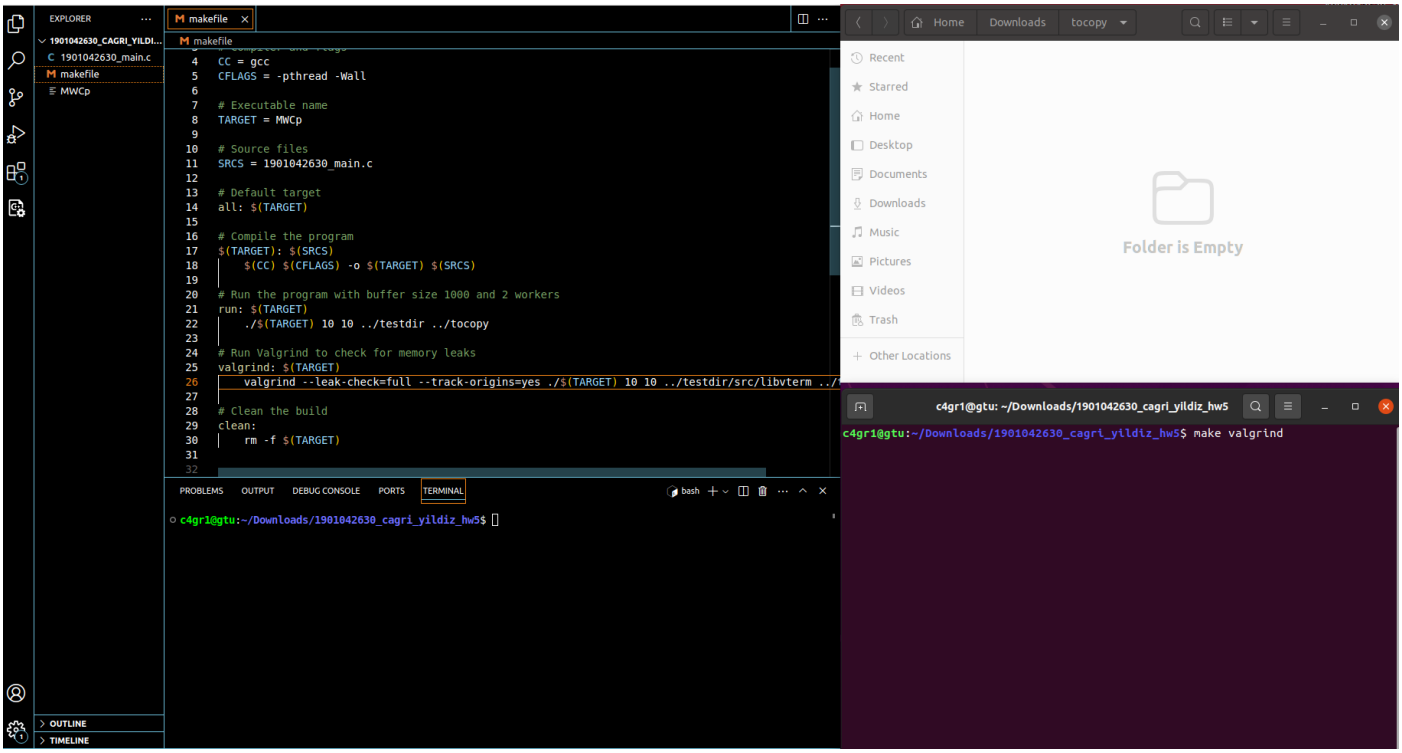


- Make valgrind (with 10 10 ../testdir ../tocopy) (CTRL + C Handling)

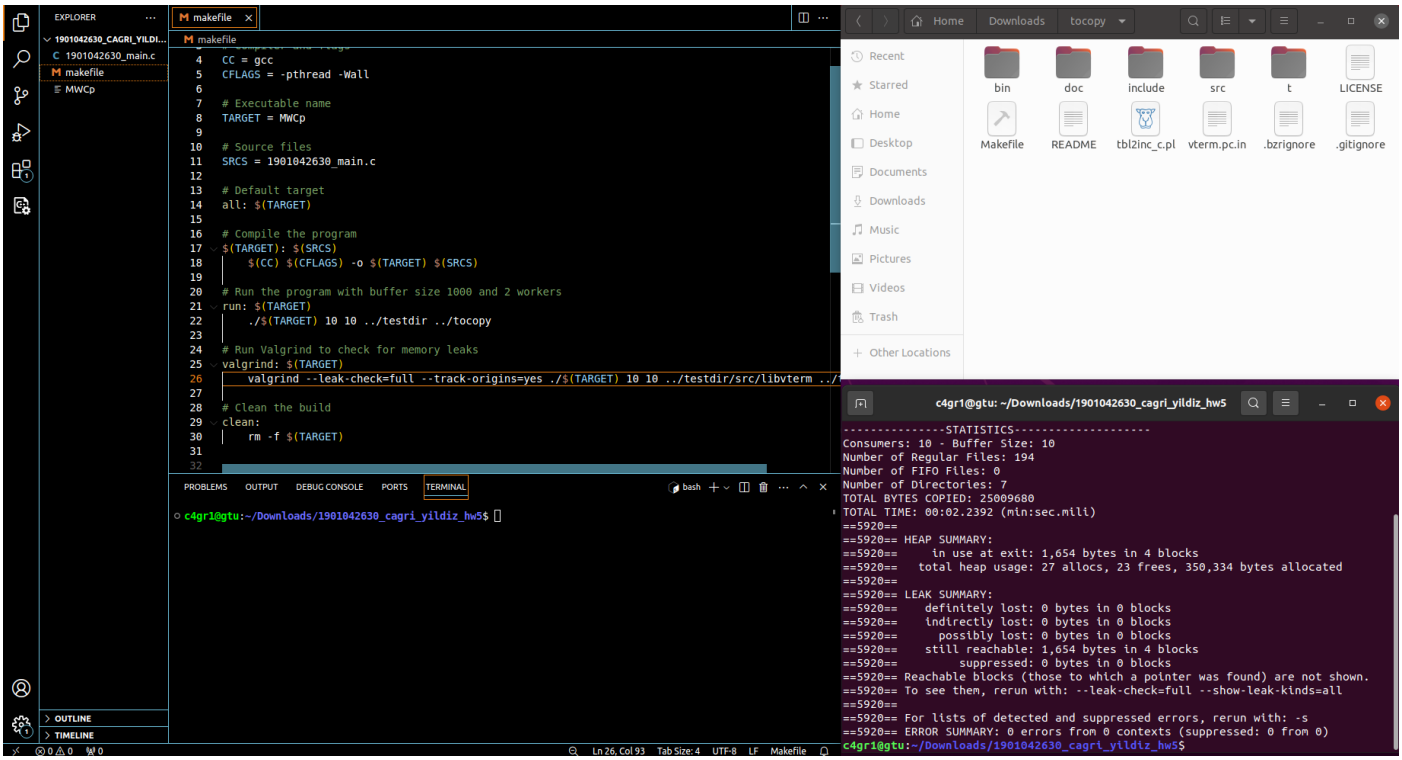


- **Test1:** valgrind ./MWCp 10 10 ../testdir/src/libvterm ../tocopy

Before command

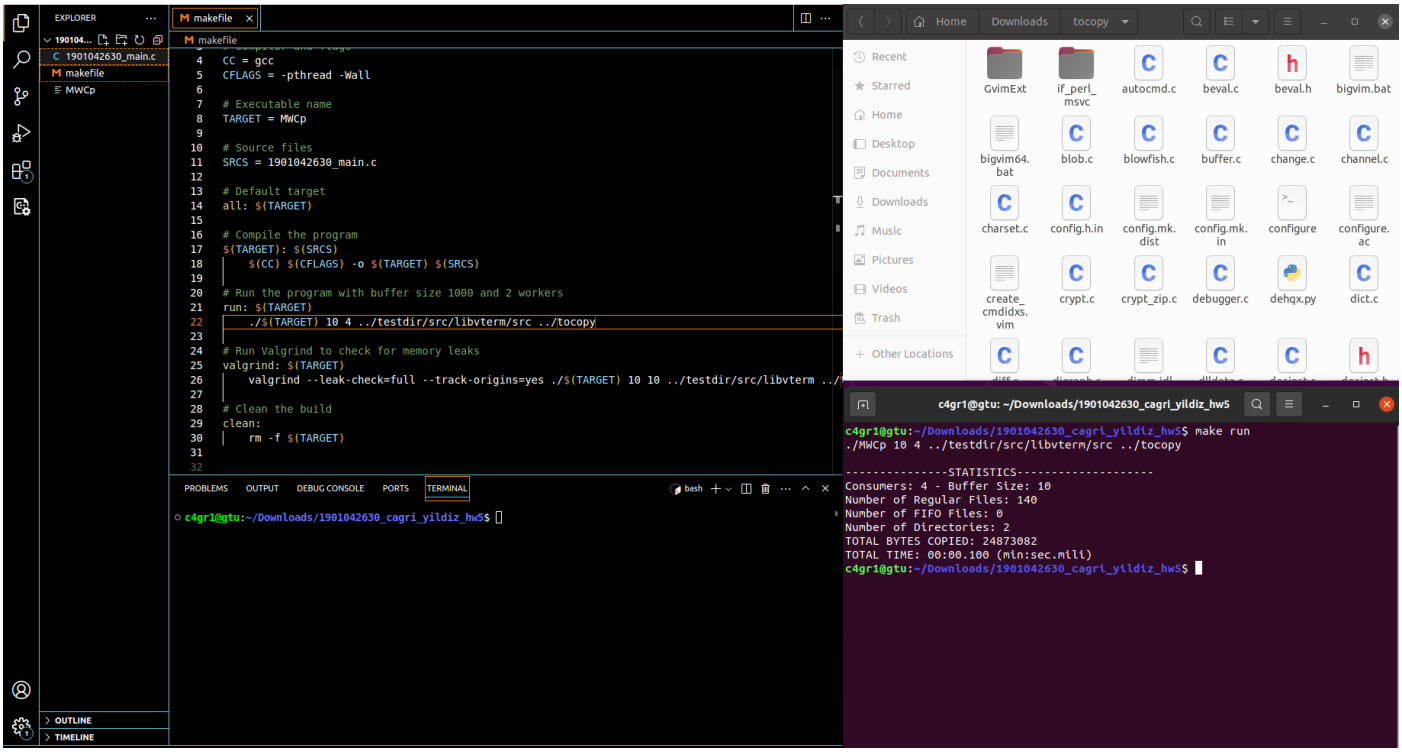


After command

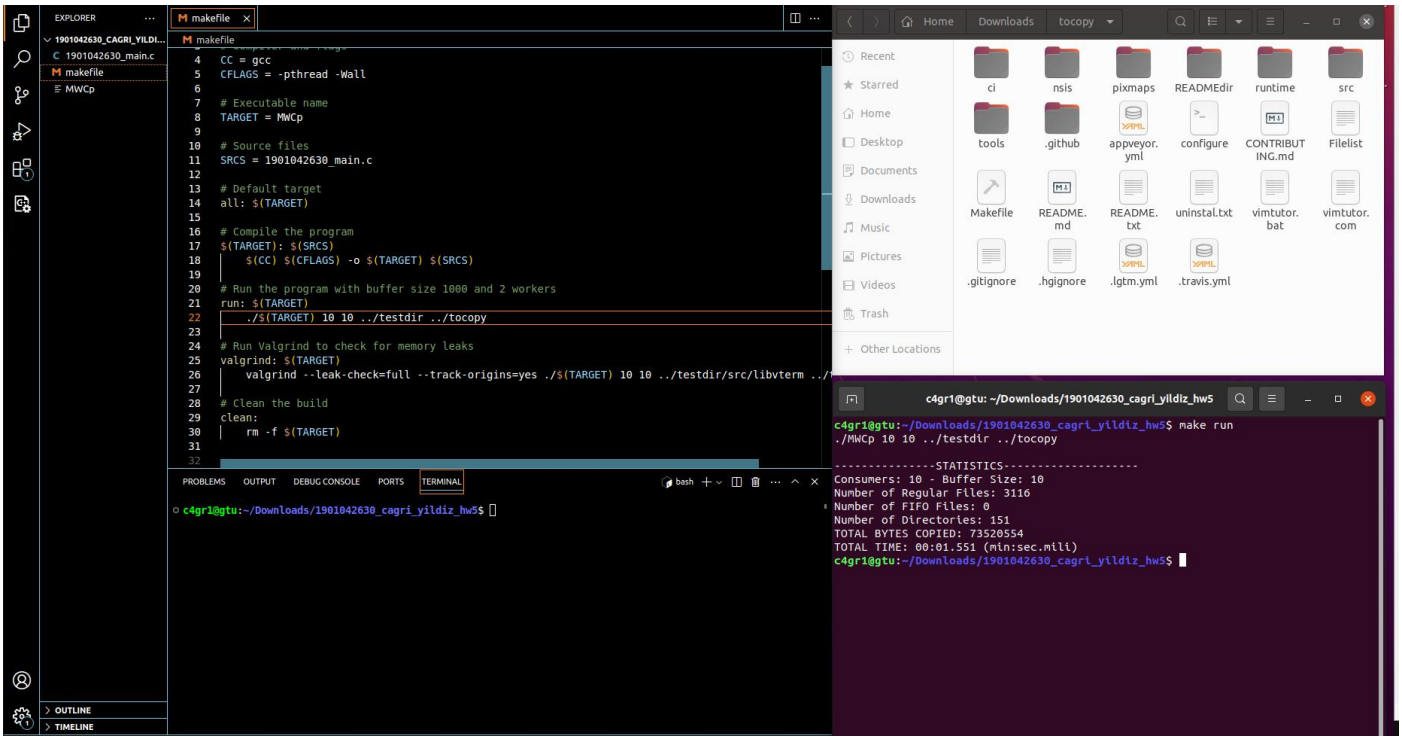


Note: I deleted the files in the tocopy folder after command.

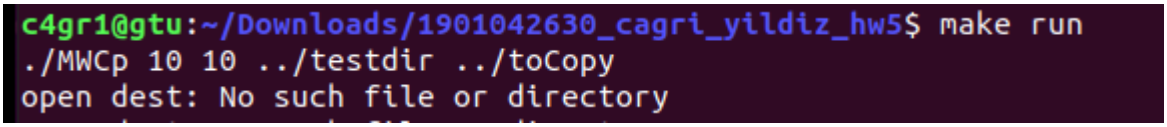
- **Test2**: ./MWCp 10 4 ../testdir/src/libvterm/src ../tocopy

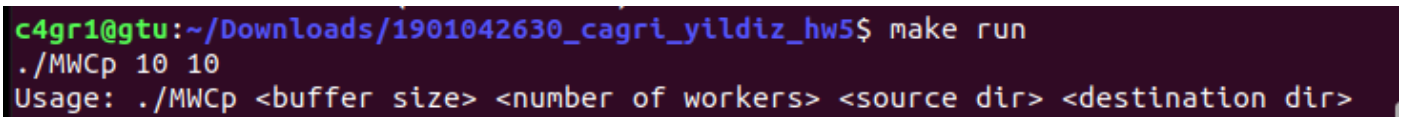


- Test3: ./MWCp 10 10 ../testdir ../toCopy



- Invalid commands





THANKS FOR READING

ÇAĞRI YILDIZ

1901042630