CSE-344 SYSTEM PROGRRAMING

SPRING 2024

HOMEWORK 5

REPORT

ONUR ATASEVER

210104004087

This homework and report is modified according to barriers. In the worker threads, int the beginnin barrier is used

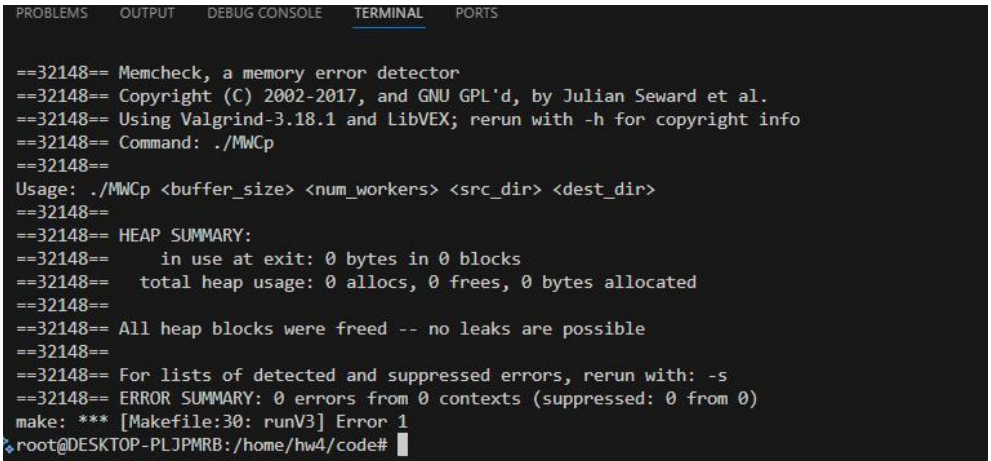
I applied worker manager approach for threads as requested. I created the Manager, the desired number of threads and the desired size buffer. The manager calls the copy_directory function and adds it to the buffer. Since the place where the buffer is added is a region that can be accessed by all threads, it is considered a critical region and is protected with a mutex. The buffer size was checked with each addition to the buffer. If the buffer is full, the thread goes to sleep and continues to block until it receives the buffer is not full signal. When the buffer is not full, every time an element is added, the buffer is not empty signal is sent. Thus, if there is a worker waiting by blocking it because the buffer is empty, it is woken up. If what is wanted to be written to the buffer is a folder rather than a file, this folder is recursively sent back to the copy_directory function.

First thread is at the beginning of the Worker threads. So all the other threads waits the other threads arrive that point. Worker threads check the buffer size when they start running. If the buffer is empty, they are blocked and start waiting. After receiving the Buffer is not empty signal, it wakes up and continues to do its job. Retrieves data from the buffer and deletes it. According to the information they receive from Buffer, they determine the file type and increase the necessary counters. Mutex protections are used in every operation regarding the buffer as it is a critical area and to provide synchronization. Then, copying is done from the source path to the destination path. It is copying byte by byte. So with valgrind it takes time to run. Second barrier is in the end of the worker function. So all the workers wait in the end before they return.

Signal handler created for CTRL+C. Each thread runs when the running variable is 1. If this signal comes, the running variable becomes 0 and the threads stop running. In addition, blocked threads are woken up in the signal handler so that the program reaches the end properly, performs the freeing operations and ends.

To run test 1 with valgrind you can use “make valgrind1” command, for test 2 “make valgrind2” and for test 3 “make valgrind3”.

For each read, write, open system calls for file, error check is done. For opening directory or getting file status error checks are done. For sigaction function, error checks is done and fort he command line arguments error check is done.



```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

==32148== Memcheck, a memory error detector
==32148== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==32148== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==32148== Command: ./MWCp
==32148==
Usage: ./MWCp <buffer_size> <num_workers> <src_dir> <dest_dir>
==32148==
==32148== HEAP SUMMARY:
==32148==     in use at exit: 0 bytes in 0 blocks
==32148==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==32148==
==32148== All heap blocks were freed -- no leaks are possible
==32148==
==32148== For lists of detected and suppressed errors, rerun with: -s
==32148== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
make: *** [Makefile:30: runV3] Error 1
root@DESKTOP-PLJPMRB:/home/hw4/code#
```

Pseudo Code For Main:

```
FUNCTION main(argc, argv)
  IF argc != 5 THEN
    PRINT "Usage: %s <buffer_size> <num_workers> <src_dir> <dest_dir>"
    EXIT_FAILURE
  SETUP signal handler for SIGINT to call intHandler
  PARSE command line arguments to get buffer_size, num_workers, src_dir, dest_dir
  ALLOCATE memory for worker threads
  INITIALIZE buffer with buffer_size
  GET current time as start time
  CREATE manager thread to run manager
  function FOR each worker thread
    CREATE worker thread to run worker function
  JOIN manager thread
  FOR each worker thread
    JOIN worker thread
  GET current time as end time
  CALCULATE execution time
  PRINT statistics: number of files, directories, total bytes copied, execution time
  FREE allocated memory for worker threads
  DESTROY buffer
  RETURN 0
END FUNCTION
```

Pseudo Code For Manager:

```
FUNCTION manager(arg)
  CALL copy_directory(src_dir, dest_dir)
  LOCK buffer.mutex
  SET buffer.done to 1
  BROADCAST buffer.not_empty // Wake up all worker threads
  UNLOCK buffer.mutex
  RETURN NULL
END FUNCTION
```

Pseudo Code For copy_directory:

```
FUNCTION copy_directory(src, dest)
  OPEN source directory
  CREATE destination directory
  WHILE there are more entries in the source directory AND running is
    TRUE IF entry is not "." or ".." THEN
      CONSTRUCT src_path and dest_path
      GET file status
      IF file is a regular file THEN
        ADD file pair to buffer
   ELSE IF file is a directory THEN
      INCREMENT directory count
        RECURSIVELY call copy_directory(src_path, dest_path)
  CLOSE source directory
END FUNCTION
```

Pseudo Code For Worker:

```
FUNCTION worker(arg)
  WHILE running IS TRUE
    REMOVE file pair from buffer
    IF file pair is empty THEN
      BREAK
    OPEN source and destination files
    COPY file contents from source to destination
    CLOSE source and destination files
  RETURN NULL
END FUNCTION
```

Mutex Protection And Condtional wait Examples:

```c
// Add a file pair to the buffer
void add_to_buffer(Buffer *buffer, const char *src, const char *dest)
{
    // printf("Locktan önce\n");
    pthread_mutex_lock(&buffer->mutex); // Lock the buffer
    // printf("Locktan sonra\n");
    while (buffer->count == buffer->buffer_size && running)
    {
        pthread_cond_wait(&buffer->not_full, &buffer->mutex); // If the buffer is full, wait for it to be not full
        if (!running) // If the program is not running, unlock the mutex to prevent other mutexes wait and return
        {
            pthread_mutex_unlock(&buffer->mutex);
            return;
        }

    }
    // printf("While'dan sonra\n");
    strncpy(buffer->buffer[buffer->tail].src, src, MAX_FILENAME_LENGTH);
    strncpy(buffer->buffer[buffer->tail].dest, dest, MAX_FILENAME_LENGTH);
    buffer->tail = (buffer->tail + 1) % buffer->buffer_size; // Update the tail
    buffer->count++; // Increment the count
    pthread_cond_signal(&buffer->not_empty); // Signal that the buffer is not empty
    pthread_mutex_unlock(&buffer->mutex); // Unlock the buffer
}
```

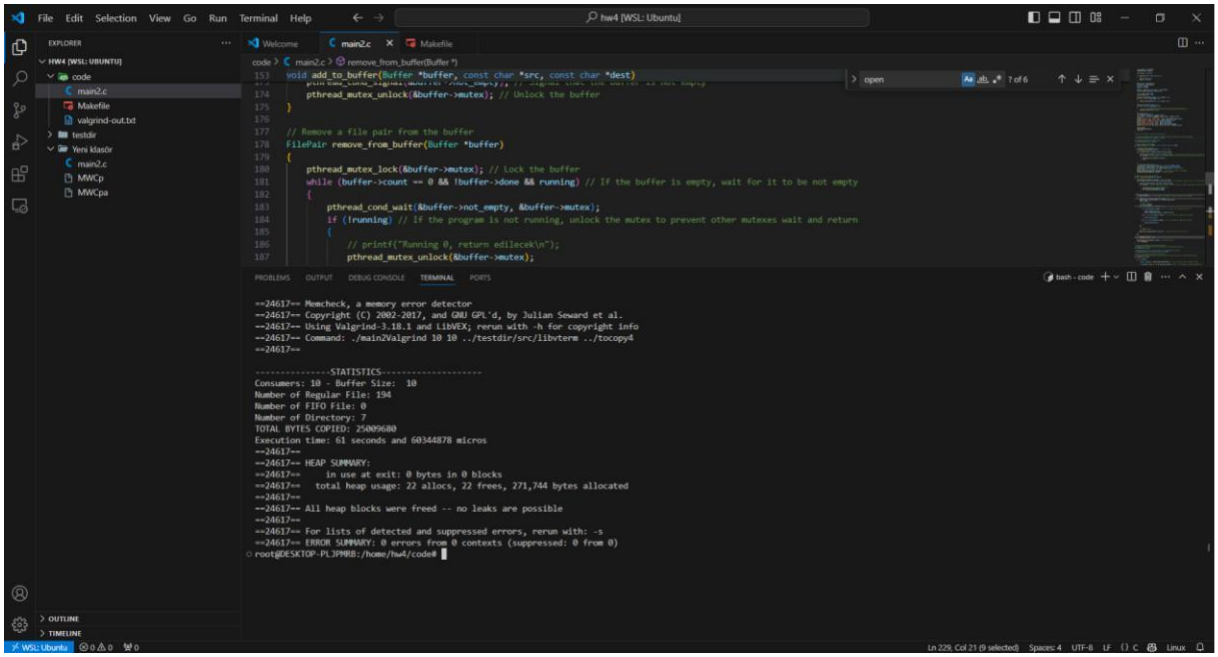
```c
// Remove a file pair from the buffer
FilePair remove_from_buffer(Buffer *buffer)
{
    pthread_mutex_lock(&buffer->mutex); // Lock the buffer
    while (buffer->count == 0 && !buffer->done && running) // If the buffer is empty, wait for it to be not empty
    {
        pthread_cond_wait(&buffer->not_empty, &buffer->mutex);
        if (!running) // If the program is not running, unlock the mutex to prevent other mutexes wait and return
        {
            // printf("Running 0, return edilecek\n");
            pthread_mutex_unlock(&buffer->mutex);
            return;
        }

    }
```

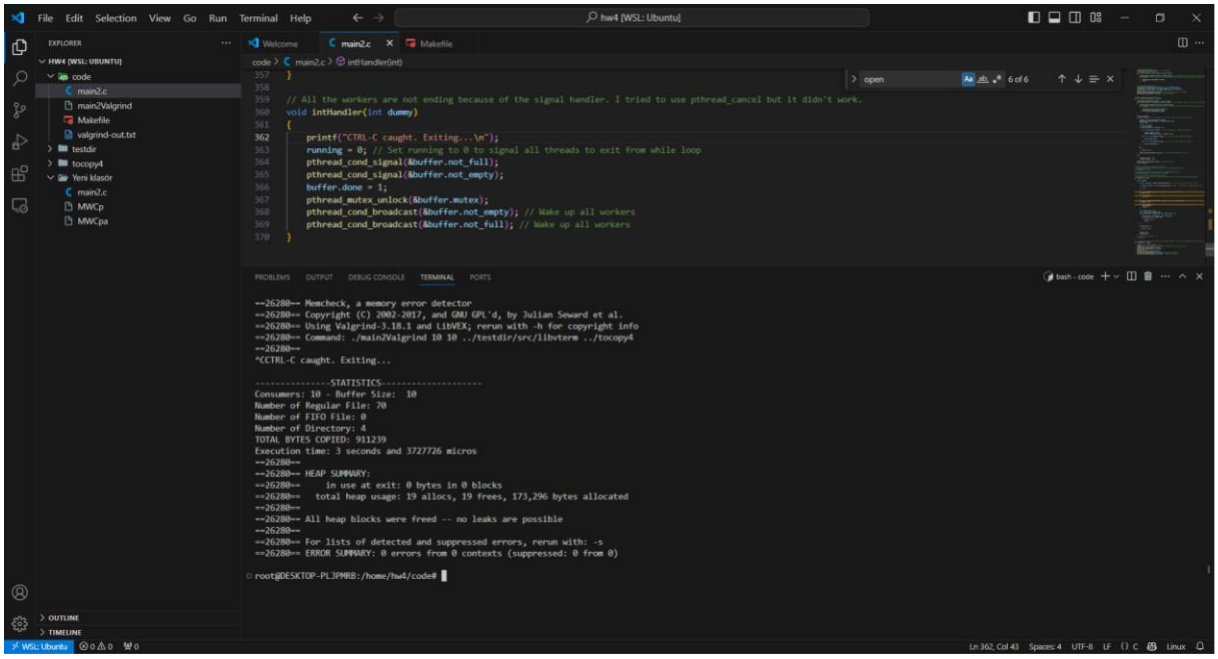
Signal Handler Example For CTRL+C:

```c
// All the workers are not ending because of the signal handler. I tried to use pthread_cancel but it didn't work.
void intHandler(int dummy)
{
    running = 0; // Set running to 0 to signal all threads to exit from while loop
    pthread_cond_signal(&buffer.not_full);
    pthread_cond_signal(&buffer.not_empty);
    buffer.done = 1;
    pthread_mutex_unlock(&buffer.mutex);
    pthread_cond_broadcast(&buffer.not_empty); // Wake up all workers
    pthread_cond_broadcast(&buffer.not_full); // Wake up all workers
}
```

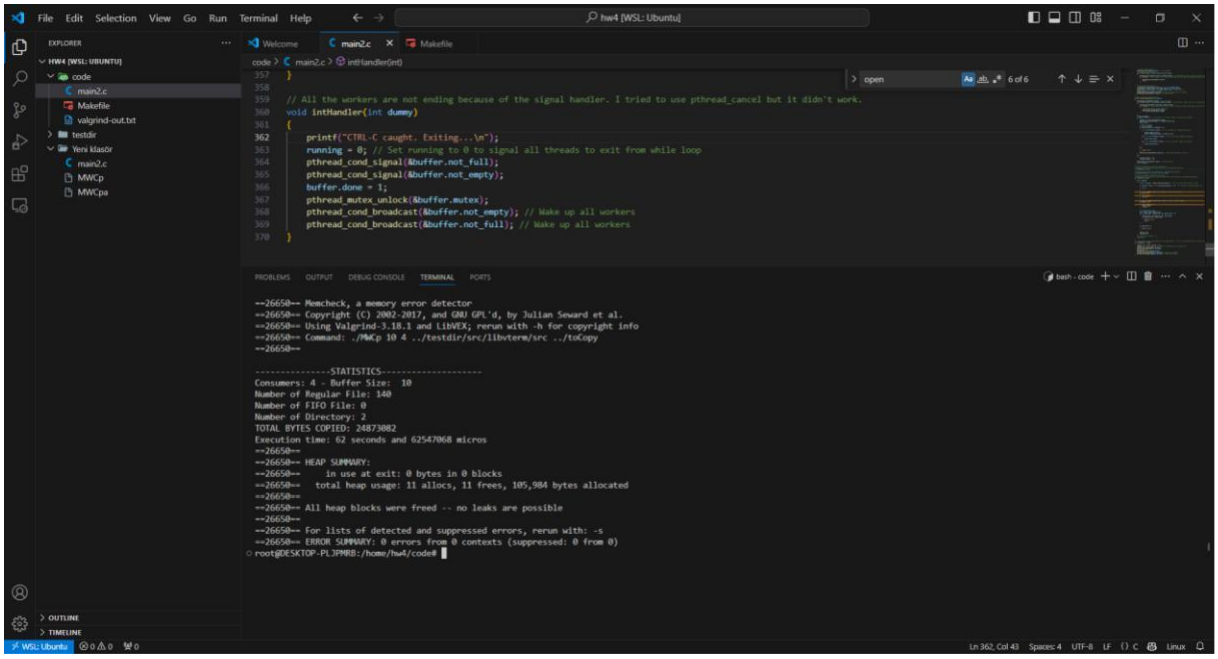
Test 1 Output:



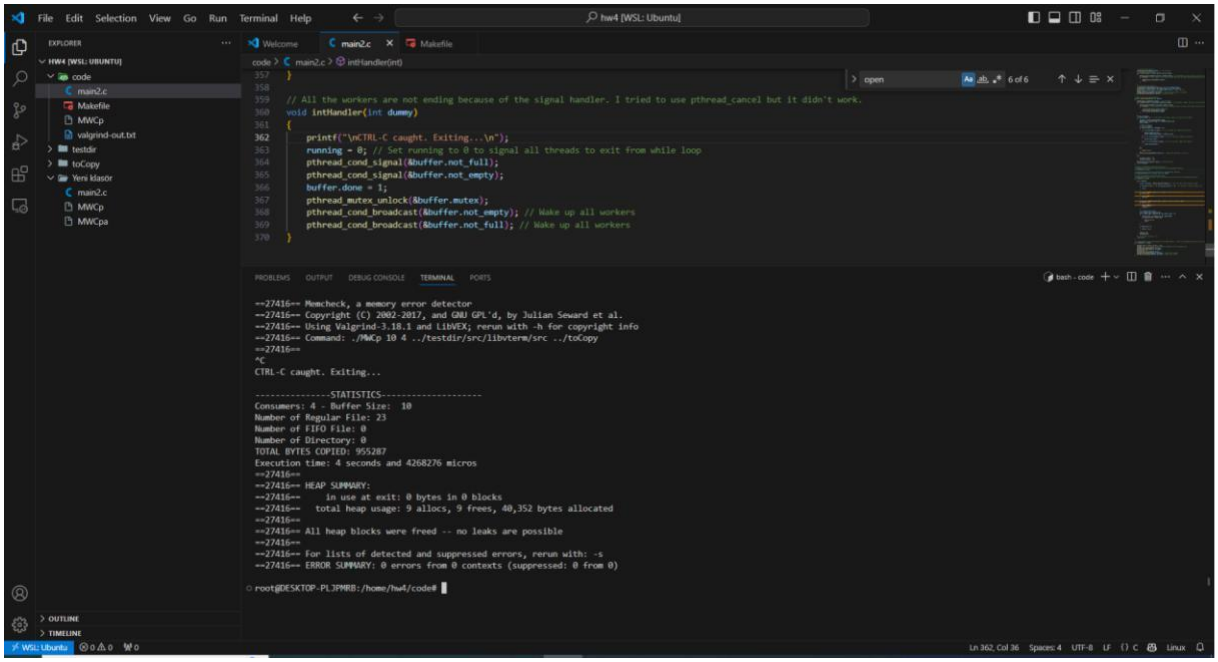
Test 1 Output With CTRL+C:



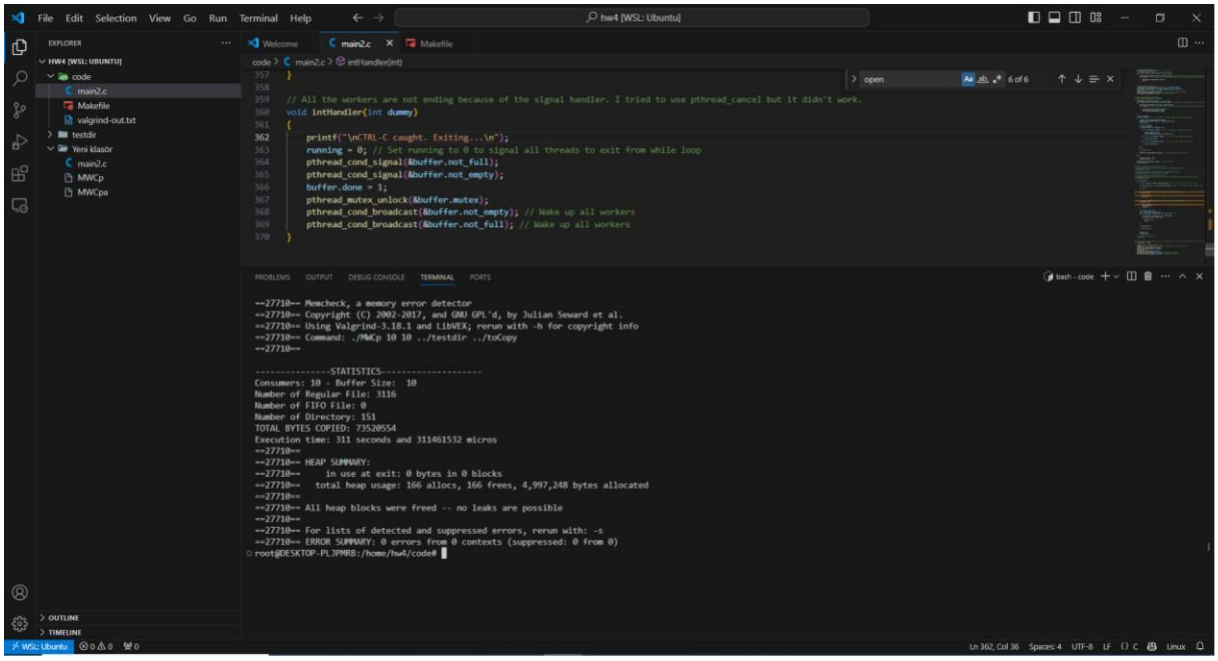
Test 2 Output:



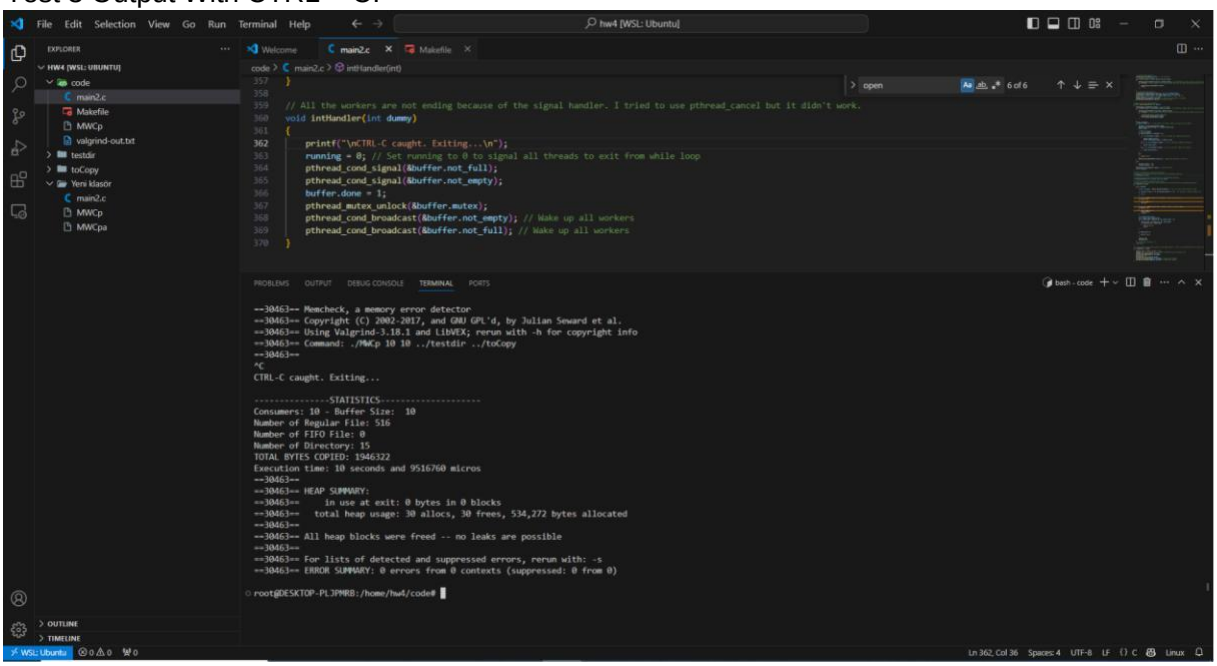
Test 2 Output With CTRL+C:



Test 3 Output:



Test 3 Output With CTRL + C:



Barrier Usage:

```c
// Worker function to copy files it gets from the buffer and write them to the destination directory
void *worker(void *arg)
{
    pthread_barrier_wait(&barrier); // Synchronize threads at this point to start copying files
    while (running)
    {
        FilePair file_pair = remove_from_buffer(&buffer); // Get a file pair from the buffer to copy

        if (file_pair.src[0] == '\0' && file_pair.dest[0] == '\0') // If the buffer is empty and done, exit
        {
```

```c
        printf("debug\n");
    }
    pthread_barrier_wait(&barrier); // Synchronize threads at this point to finish copying files
    printf("Worker exiting...\n");
    return NULL;
}
```

With this usage all the worker threads starts at the same time and they ends at the same time. This provides synchronization.