


Gebze Technical University
CSE 344
SYSTEM PROGRAMMING
HOMEWORK #4 and #5

200104004066 - Ahmet Yiğit

Important Note:
HOMEWORK OF 4 AND 5 SOURCE FILES
ARE SAME. BARRIERS AND CONDITION
VARIABLES ARE USED.

Short Introduction to Functions:


- **Main Function:**



```
1  int main(int argc, char *argv[]) {
2      ParseCommandLineArguments(argc, argv);
3
4      InitializeResources();
5
6      CreateThreads();
7
8      WaitForThreads();
9
10     PrintStatistics();
11
12     CleanupResources();
13
14     return 0;
15 }
16
```

- The main function orchestrates the program's execution by parsing command-line arguments, initializing resources, creating threads, waiting for threads to finish, printing statistics, and cleaning up resources before exiting.

- **Manager - Manager Helper**



```
1  void *manager(void *args) {
2      ThreadArgs *threadArgs = (ThreadArgs *)args;
3      char *srcDir = threadArgs->srcDir;
4      char *destDir = threadArgs->destDir;
5
6      start_time = clock();
7
8      // Begin recursive traversal of source directory
9      managerHelper(srcDir, destDir);
10
11     isCopyDone = 1;
12     pthread_cond_broadcast(&buffer.empty);
13
14     pthread_barrier_wait(&barrier);
15
16     free(args);
17
18     end_time = clock();
19
20     pthread_exit(NULL);
21 }
```

- The manager function coordinates the file copying tasks. It starts a timer, traverses the source directory to enqueue files for copying, signals when

copying is done, waits for worker threads to finish copying, stops the timer, and then exits.

```
1 void managerHelper(const char *srcDir, const char *destDir)
2 // Open source directory
3 DIR *dir = opendir(srcDir);
4
5 if (dir == NULL) {
6     perror("Error opening source directory");
7     return;
8 }
9
10 // Loop through directory entries
11 while ((entry = readdir(dir)) != NULL) {
12     // If entry is a directory
13     if (isDirectory(entry)) {
14         createDirectory(entry);
15         // recursive to get all files in the dir
16         managerHelper(entry, destDir);
17     } else {
18         produceFileForCopying(entry, destDir); // produce
19         updateStatistics(entry);
20     }
21 }
22 // Close directory
23 closedir(dir);
24 }
25
```

- The managerHelper function is responsible for recursively traversing the source directory. It iterates through each entry in the directory, distinguishing between directories and files. For directories, it creates the corresponding directory in the destination path and recursively calls itself for the subdirectory. For files, it prepares file data for copying and enqueues it for copying, while also updating relevant statistics.

- Produce

```
1 void produce(FileData data) {
2     lockMutex(buffer.mutex);
3     while (buffer.count >= bufferSize) {
4         waitCondition(buffer.full, buffer.mutex);
5     }
6     if (isCopyDone) {
7         unlockMutex(buffer.mutex);
8         return;
9     }
10    addToBuffer(data);
11    signalCondition(buffer.empty);
12    unlockMutex(buffer.mutex);
13 }
```

- The produce function locks the buffer mutex and waits while the buffer is full. If copying is done, it unlocks the mutex and returns. Otherwise, it adds file data to the buffer, signals that the buffer is not empty, and unlocks the mutex.

- CreateDirectory

- It creates a directory in the destination folder.

- **Worker**

```
1 void *worker(void *args) {
2     while (!isCopyDone) {
3         FileData data = consume();
4         if (strlen(data.srcName) == 0) break;
5         copyFile(data);
6         printCopiedFileInfo(data);
7     }
8     barrierWait();
9     return NULL;
10 }
```

-
- The worker function continuously consumes file data from the buffer until copying is done. For each file data consumed, it copies the file from source to destination and prints information about the copied file. After all copying is completed, it waits for other worker threads to finish using a barrier synchronization mechanism.

- **Consume**

```
1 FileData consume() {
2     lockMutex(buffer.mutex);
3     while (buffer.count <= 0 && !isCopyDone) {
4         waitCondition(buffer.empty, buffer.mutex);
5     }
6     if (buffer.count <= 0 && isCopyDone) {
7         unlockMutex(buffer.mutex);
8         return emptyFileData();
9     }
10    FileData data = removeFromBuffer();
11    signalCondition(buffer.full);
12    unlockMutex(buffer.mutex);
13    return data;
14 }
```

-
- The consume function retrieves file data from the buffer for processing by worker threads. It locks the buffer mutex, waits while the buffer is empty (and copying is not yet done), retrieves file data from the buffer, signals that the buffer is not full, and finally unlocks the mutex before returning the file data.

- **CopyFile**

- It copies the file given source path to destination path.

- **HandleSignal**

- It handles CTRL+C signals, when CTRL+C comes, it consumes all the buffer and does not take any new information to the buffer, finishing all the threads.

- **DeleteDirectoryContent**

- Delete directory and its folder/files recursively.

- **Condition Variables**

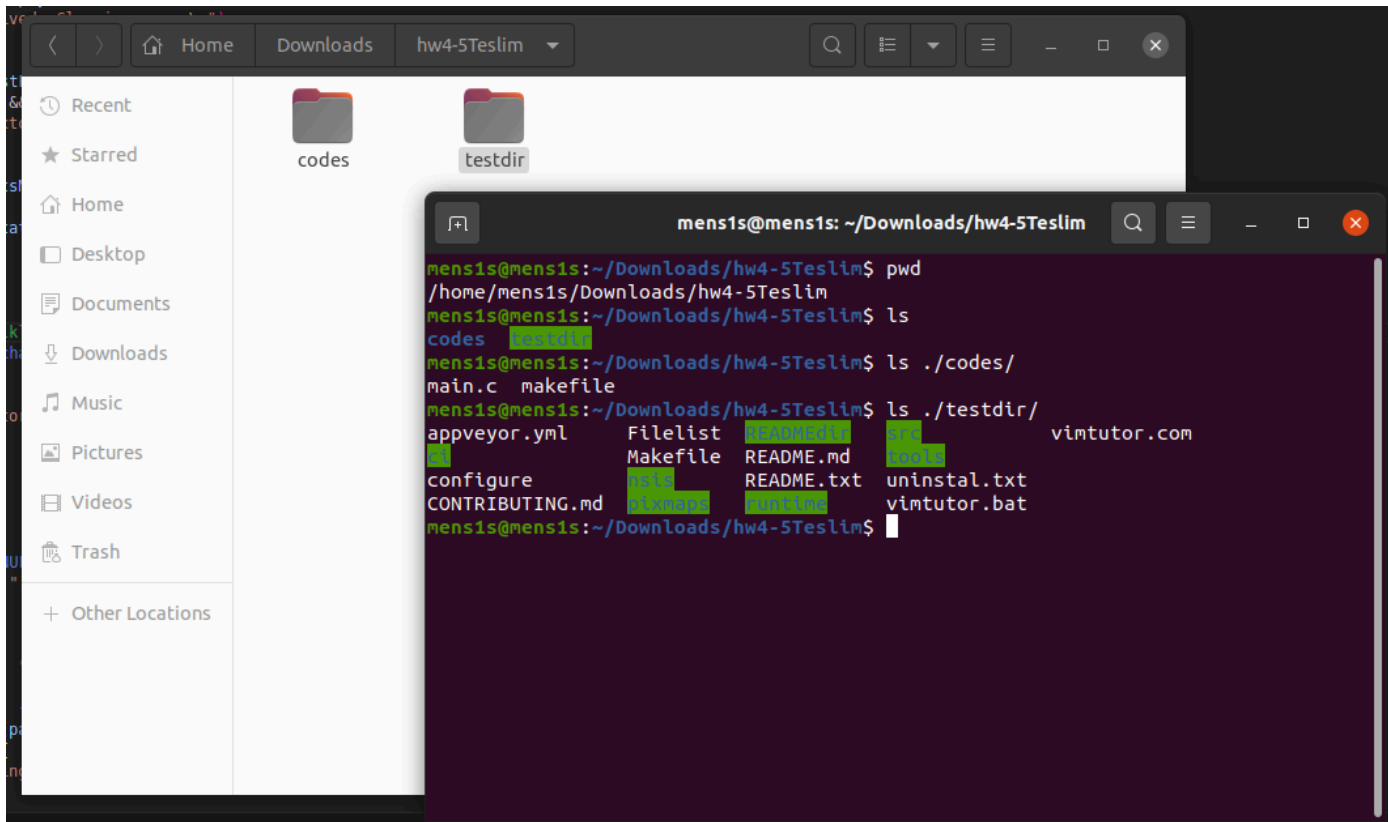
- Buffer Full Condition (`buffer.full`)
- Buffer Empty Condition (`buffer.empty`)
- In Produce Function
 - The produce function locks the mutex associated with the buffer.
 - It checks if the buffer is full. If so, it waits on the `buffer.full` condition, releasing the mutex and sleeping until the condition is signaled.
- In Consume Function
 - The consume function locks the mutex associated with the buffer.
 - It checks if the buffer is empty and if copying is not done. If so, it waits on the `buffer.empty` condition, releasing the mutex and sleeping until the condition is signaled.
 - If the buffer is still empty and copying is done, it unlocks the mutex and returns an empty `FileData` structure.
 - If there is data in the buffer, it removes the data from the buffer.
 - It then signals the `buffer.full` condition to wake up any waiting producer threads, indicating that there is now space available in the buffer.
 - Finally, it unlocks the mutex and returns the data.

- **Barriers**

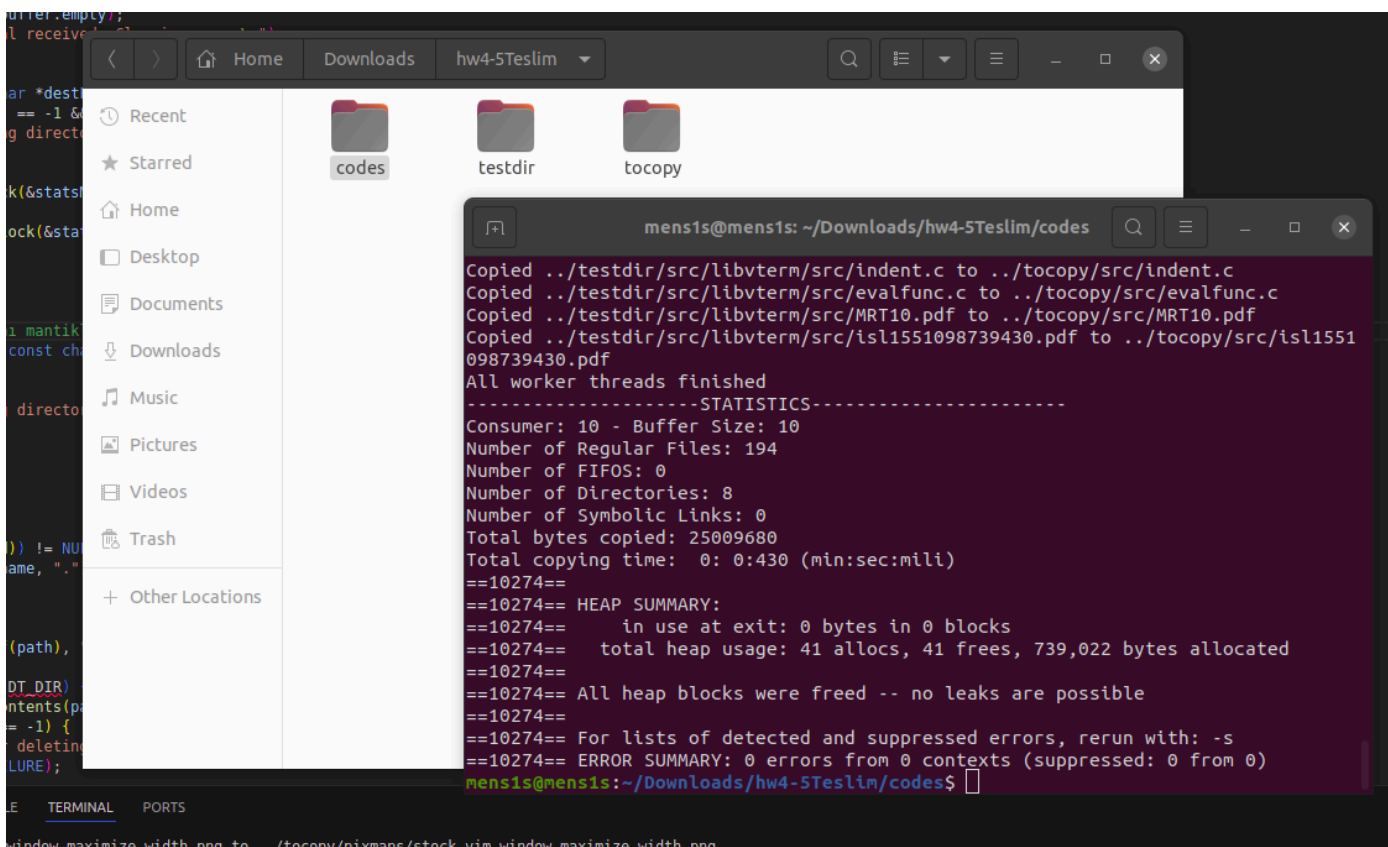
- End of Manager Function
 - To ensure the manager thread waits until all worker threads have finished copying the files before it exits and finalizes the process.
 - The manager function starts by traversing the source directory and enqueueing files for copying.
 - After enqueueing all files, it sets a flag to signal that copying is done.
 - It then waits for all worker threads to finish copying files by calling `pthread_barrier_wait(&barrier)`.
 - This call blocks the manager thread until all worker threads also reach their corresponding barrier point, ensuring all threads synchronize at this point before the manager thread proceeds to stop the timer and exit.
- End of Worker Function
 - To ensure that each worker thread waits for all other worker threads to finish their copying tasks before any of them proceed to the next step or exit.
 - The worker function runs in a loop, consuming file data and copying files until copying is done.
 - After completing their tasks, each worker thread waits for all other worker threads to finish by calling `pthread_barrier_wait(&bar)`.
 - This call blocks the worker thread until all threads, including the manager thread, reach the barrier point.
 - This ensures that no worker thread exits or moves to the next stage until all threads have finished their copying tasks.

- TEST CASES

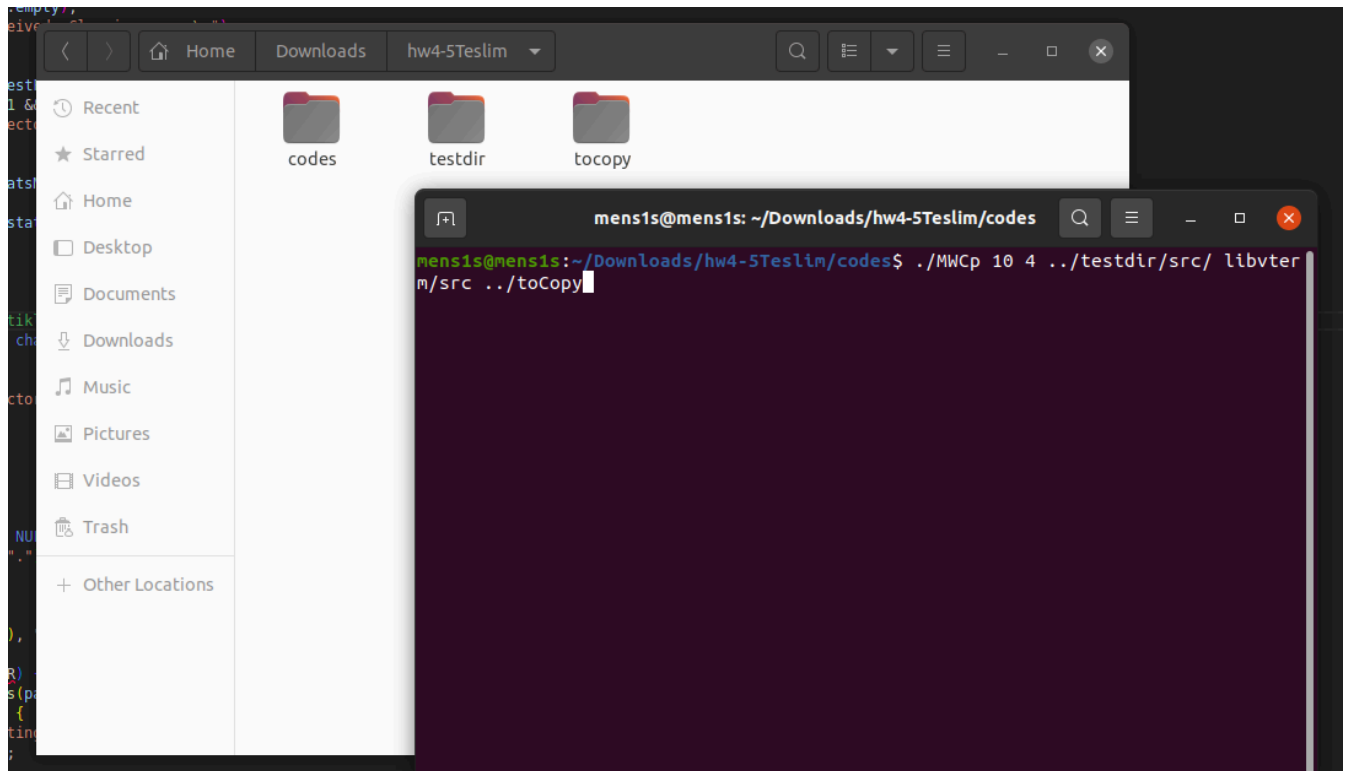
- valgrind ./MWCp 10 10 ../testdir/src/libvterm ../tocopy
- Init State



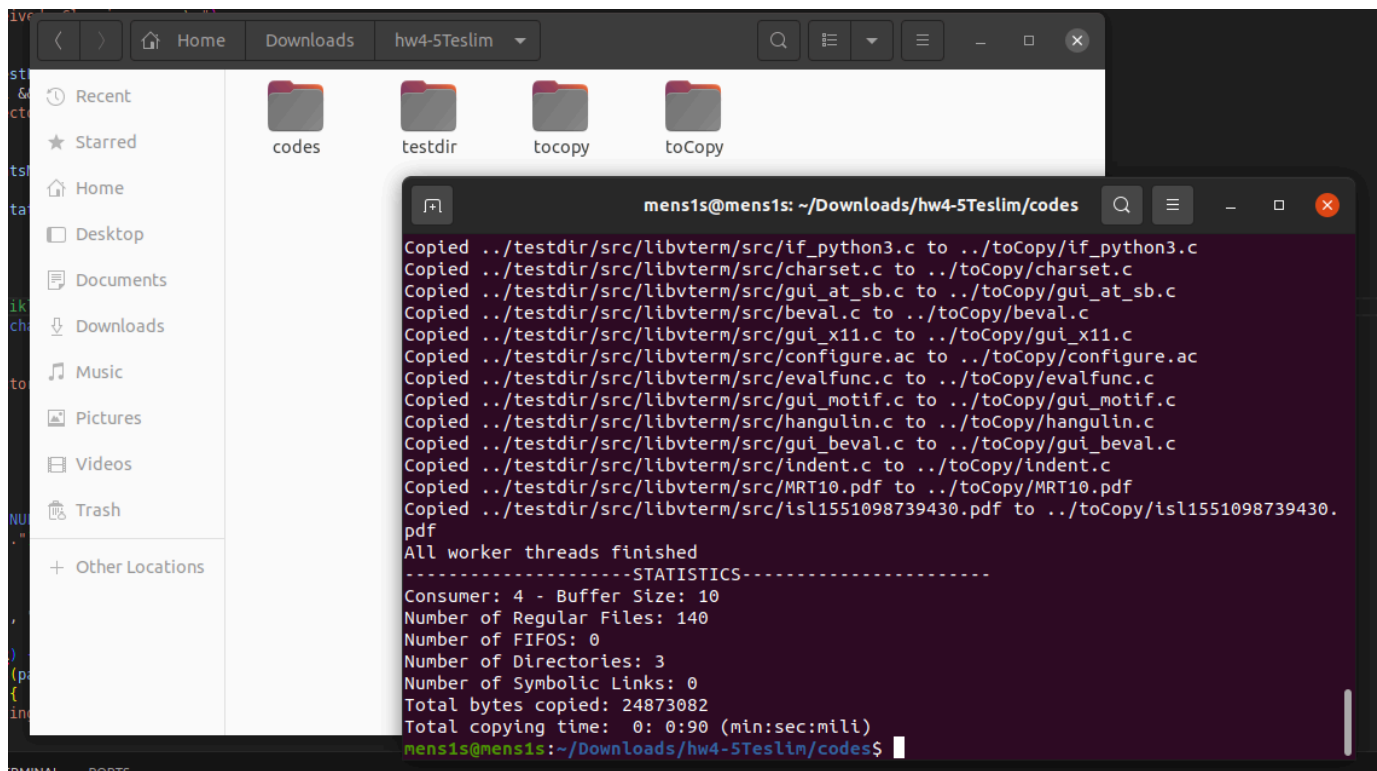
- Final State



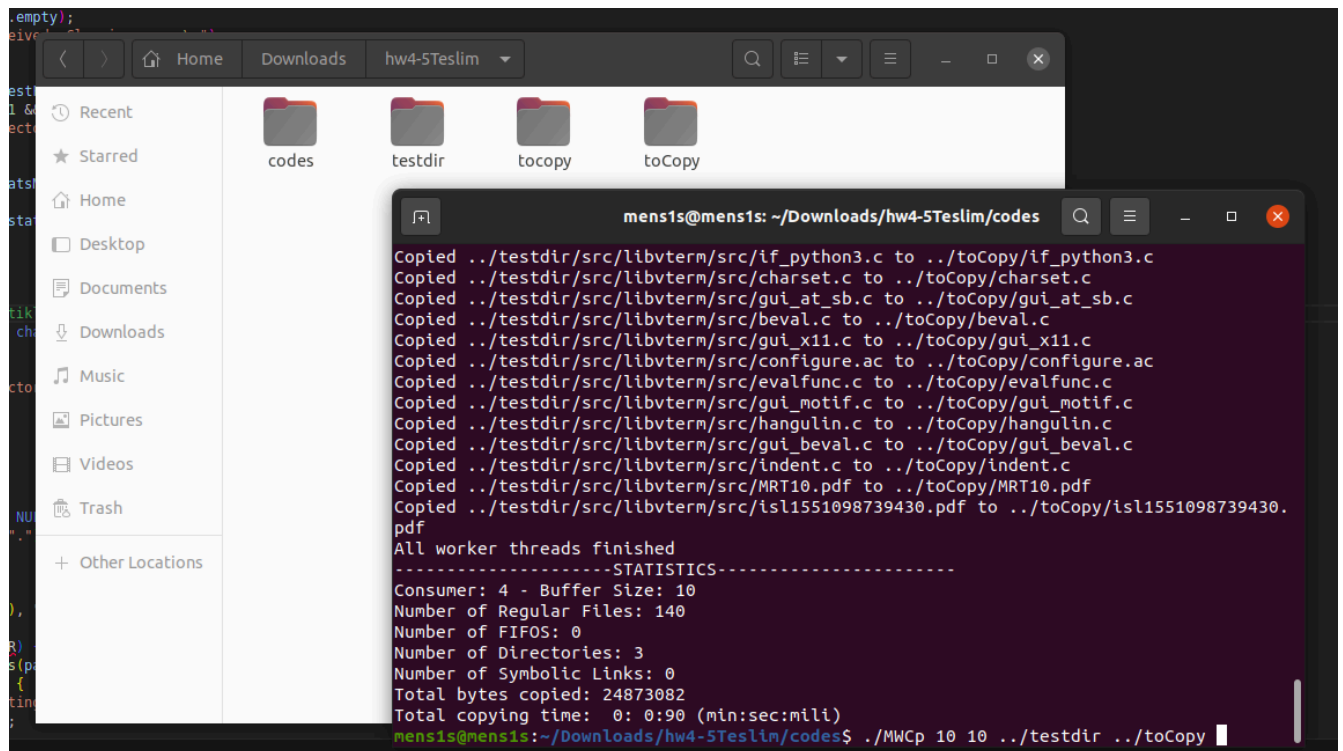
- `./MWCp 10 4 ../testdir/src/libvterm/src ../toCopy` create toCopy
- Init State



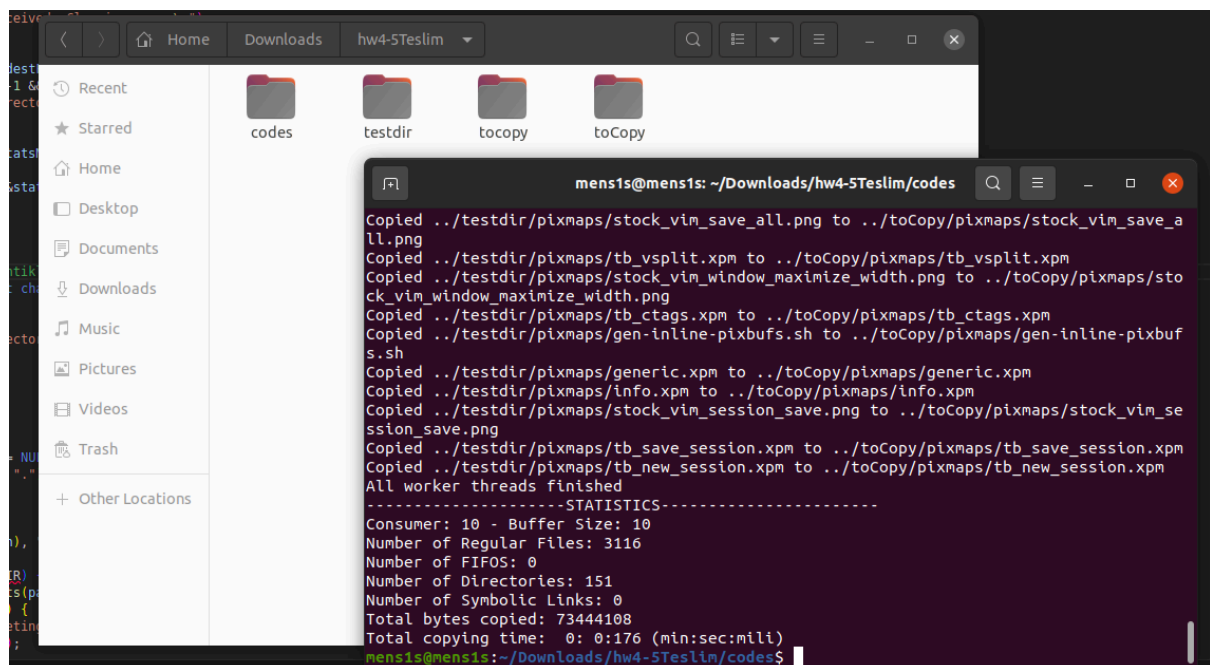
- Final State



- `./MWCp 10 10 ../testdir ../toCopy`
- Init State

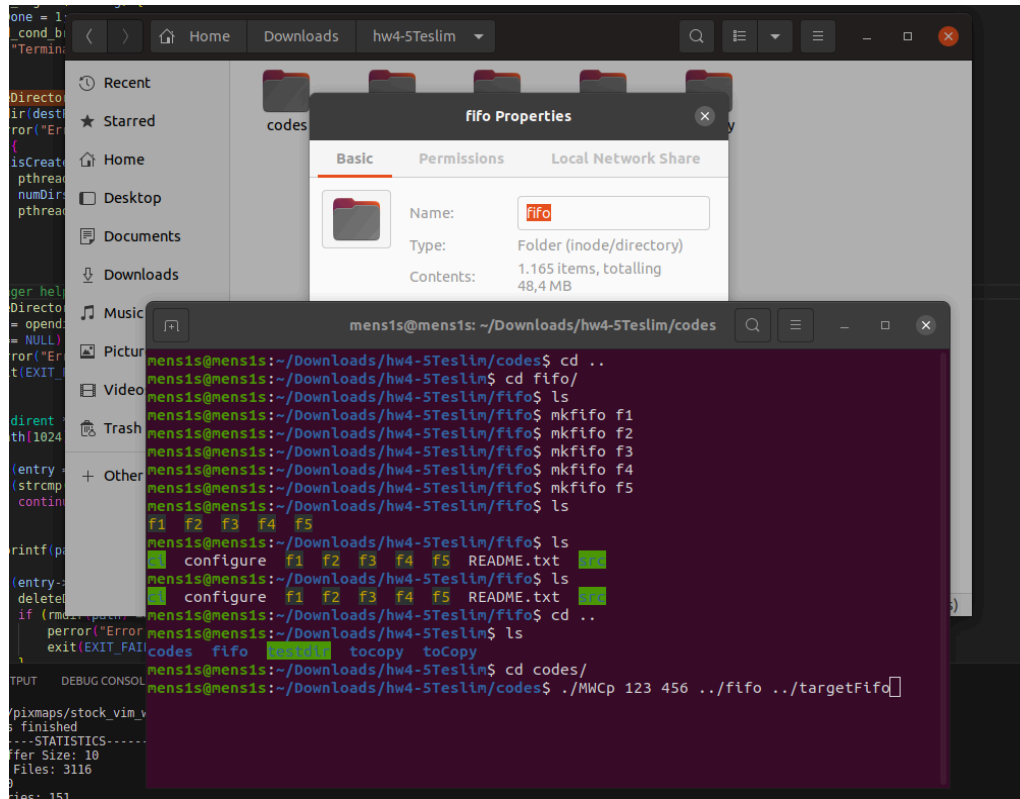


- Final State



FIFO AND OTHER FILE COPY PROCESS

- Init State



- Final State

