CSE – 344 Homework 4 Report

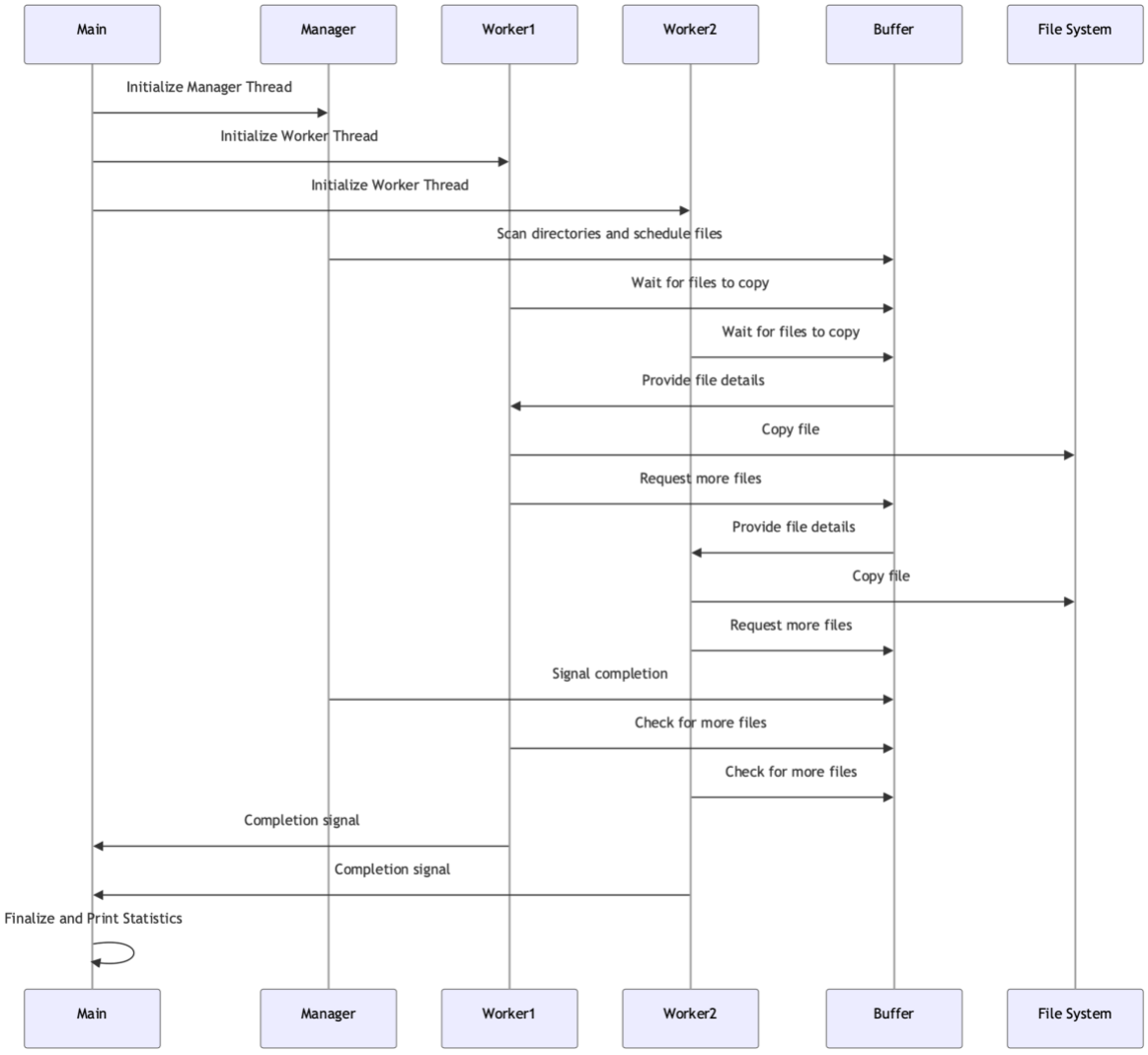
Süleyman Burak Yaşar
1901042662

Table of Contents

Usage Of Makefile

make: Compiles the code
run_test: valgrind ./MWCp 10 10 ../testdir/src/libvterm ../tocopy
run_test2: ./MWCp 10 4 ../testdir/src/ libvterm/src ../toCopy
run_test3: ./MWCp 10 10 ../testdir ../toCopy
clean: Remove MWCp

Design and structure of the program



File Buffer Structure

Buffer

- **buffer**: An array of **FilePair** structures. This array contains the file pairs that will be stored in the buffer.
- **MAX_BUFFER_SIZE**: This defines the maximum capacity of the buffer. For example, if set to 100, the buffer can hold up to 100 file pairs.
- **buffer_size**: An integer representing the size of the buffer. This value is provided as a command-line argument when running the program.
- **in**: The index where a new item will be added to the buffer. This index points to the position where new file pairs will be inserted into the buffer. It increments cyclically.
- **out**: The index where an item will be removed from the buffer. This index points to the position from which file pairs will be taken out of the buffer. It increments cyclically.
- **count**: An integer that keeps track of the current number of items in the buffer. The **volatile** keyword indicates that this variable can be modified by different threads and should be read fresh each time.

```c
#define MAX_PATH 4096
#define MAX_BUFFER_SIZE 100

typedef struct {
    char src_path[MAX_PATH];
    char dst_path[MAX_PATH];
} FilePair;

typedef struct {
    FilePair buffer[MAX_BUFFER_SIZE];
    int buffer_size;
    int in;
    int out;
    volatile int count;
    volatile int done;
    pthread_mutex_t mutex;
    pthread_cond_t not_full;
    pthread_cond_t not_empty;
} Buffer;

Buffer buffer;
```

- **done**: A flag indicating whether the copying operation is complete. The **volatile** keyword ensures that the variable is always read from memory and not cached, as it can be changed by different threads.
- **mutex**: A mutex used to control concurrent access to the buffer. This ensures that only one thread can modify the buffer at a time, maintaining data integrity.
- **not_full**: A condition variable that indicates the buffer is not full. If the buffer is full and a thread wants to add a new item, it will wait on this condition variable.
- **not_empty**: A condition variable that indicates the buffer is not empty. If the buffer is empty and a thread wants to remove an item, it will wait on this condition variable.

The buffer is used to safely add and remove file pairs. These operations are managed through synchronized access and condition variables.

Adding a File Pair

1. **Lock**: Lock the mutex to ensure exclusive access to the buffer.
2. **Wait**: If the buffer is full (**count == buffer_size**), wait on the **not_full** condition variable.
3. **Add**: Add the file pair at the **in** index and increment **in** cyclically.
4. **Update**: Increment the **count**.
5. **Signal**: Signal the **not_empty** condition variable.
6. **Unlock**: Unlock the mutex.

Removing a File Pair

1. **Lock**: Lock the mutex to ensure exclusive access to the buffer.
2. **Wait**: If the buffer is empty (**count == 0**), wait on the **not_empty** condition variable.
3. **Remove**: Remove the file pair at the **out** index and increment **out** cyclically.
4. **Update**: Decrement the **count**.
5. **Signal**: Signal the **not_full** condition variable.
6. **Unlock**: Unlock the mutex.

Main

Argument Checking and Parsing

- The program expects exactly 4 command-line arguments: buffer size, number of worker threads, source directory, and destination directory.
- If the number of arguments is incorrect, it prints a usage message and exits.

```
if (argc != 5) {
    fprintf(stderr, "Usage: %s <buffer_size> <num_workers> <src_dir> <dst_dir>\n", argv[0]);
    exit(EXIT_FAILURE);
}
```

- **buffer_size** and **num_workers** are parsed from the command-line arguments.
- The program checks if **buffer_size** and **num_workers** are greater than 0. If not, it prints an error message and exits.

```c
int buffer_size = atoi(argv[1]);
num_workers = atoi(argv[2]);
char *src_dir = argv[3];
char *dst_dir = argv[4];

if (buffer_size <= 0 || num_workers <= 0) {
    fprintf(stderr, "Invalid buffer size or number of workers\n");
    exit(EXIT_FAILURE);
}
```

Signal Handling Setup

- Sets up a signal handler for **SIGINT** (Ctrl + C). When this signal is received, the **handle_signal** function will be called.
- This allows the program to clean up resources properly when interrupted.

```c
// Signal handler
struct sigaction sa;
sa.sa_handler = handle_signal;
sa.sa_flags = 0;
sigemptyset(&sa.sa_mask);
sigaction(SIGINT, &sa, NULL);
```

Buffer Initialization

- Initializes the buffer structure with the specified size.
- Sets the initial values for the buffer indices (**in** and **out**), the count of items in the buffer, and the done flag.
- Initializes the mutex and condition variables for synchronizing access to the buffer.

```c
buffer.buffer_size = buffer_size;
buffer.in = 0;
buffer.out = 0;
buffer.count = 0;
buffer.done = 0;
pthread_mutex_init(&buffer.mutex, NULL);
pthread_cond_init(&buffer.not_full, NULL);
pthread_cond_init(&buffer.not_empty, NULL);
```

Thread Creation

- Declares a thread for the manager and an array of threads for the workers.

```
pthread_t manager_thread;
pthread_t worker_threads[num_workers];
```

Timing the Execution

- Records the start time of the execution for performance measurement.

```
struct timespec start, end;
clock_gettime(CLOCK_REALTIME, &start);
```

Creating the Destination Directory

- Creates the destination directory if it does not already exist.

```
create_directory(dst_dir);
```

Starting the Manager Thread

- Creates the manager thread, passing the source and destination directories as arguments.

```
char *args[2] = {src_dir, dst_dir};
pthread_create(&manager_thread, NULL, manager, (void *)args);
```

Starting the Worker Threads

- Creates the worker threads. Each worker thread will execute the **worker** function.

```
for (int i = 0; i < num_workers; i++) {
    pthread_create(&worker_threads[i], NULL, worker, NULL);
}
```

Waiting for Threads to Complete

- Waits for the manager thread to complete.
- Waits for all worker threads to complete.

```c
pthread_join(manager_thread, NULL);

for (int i = 0; i < num_workers; i++) {
    pthread_join(worker_threads[i], NULL);
}
```

Measuring and Printing Execution Time

- Records the end time of the execution and calculates the elapsed time.
- Prints statistics about the number of files and directories copied, the total bytes copied, and the total execution time.

```c
clock_gettime(CLOCK_REALTIME, &end);
long seconds = end.tv_sec - start.tv_sec;
long milliseconds = (end.tv_nsec - start.tv_nsec) / 1000000;
long minutes = seconds / 60;

printf("\n----------STATISTICS----------\n");
printf("Consumers: %d - Buffer Size: %d\n", num_workers, buffer_size);
printf("Number of Regular Files: %d\n", num_files);
printf("Number of FIFO Files: %d\n", num_fifos);
printf("Number of Directories: %d\n", num_dirs);
printf("TOTAL BYTES COPIED: %d\n", num_bytes);
printf("TOTAL TIME: %02ld:%02ld.%03ld (min:sec.milli)\n", minutes, seconds, mi

pthread_mutex_destroy(&buffer.mutex);
pthread_cond_destroy(&buffer.not_full);
pthread_cond_destroy(&buffer.not_empty);
```

Cleaning Up

- Destroys the mutex and condition variables to free up resources.

```c
pthread_mutex_destroy(&buffer.mutex);
pthread_cond_destroy(&buffer.not_full);
pthread_cond_destroy(&buffer.not_empty);
```

Manager Thread

Argument Parsing

- The manager thread receives the source and destination directories as arguments.
- **src_dir** points to the source directory path.
- **dst_dir** points to the destination directory path.

```
char **dirs = (char **)arg;
char *src_dir = dirs[0];
char *dst_dir = dirs[1];
```

Copying the Directory

- Calls the **copy_directory** function to start traversing the source directory and copying its contents to the destination directory.
- The **copy_directory** function is recursive and handles both files and subdirectories.

```
copy_directory(src_dir, dst_dir, src_dir);
```

Setting the Done Flag

- After the directory traversal and copying are complete, the manager thread locks the buffer's mutex.
- Sets the buffer.done flag to 1 to indicate that all tasks have been added to the buffer.
- Broadcasts the not_empty condition variable to wake up any worker threads that might be waiting for tasks.
- Unlocks the mutex to allow other threads to access the buffer.

```
pthread_mutex_lock(&buffer.mutex);
buffer.done = 1;
pthread_cond_broadcast(&buffer.not_empty);
pthread_mutex_unlock(&buffer.mutex);
```

Copy Directory Function

The **copy_directory** function is responsible for recursively traversing the directory structure and adding tasks to the buffer for the worker threads to process.

Opening the Directory

- Attempts to open the source directory. If it fails, prints an error message and returns.

```
DIR *dir = opendir(src_dir);
if (!dir) {
    perror("opendir");
    return;
}
```

Reading Directory Entries

- Reads each entry in the directory. The **readdir** function returns **NULL** when there are no more entries.

```
struct dirent *entry;
while ((entry = readdir(dir)) != NULL) {
```

Skipping Special Entries

- Skips the **.** and **..** entries, which represent the current and parent directories, respectively.

```
if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0) {
    continue;
}
```

Constructing Source and Destination Paths

- Constructs the full path for the source and destination files or directories.

```
char src_path[MAX_PATH], dst_path[MAX_PATH];
snprintf(src_path, MAX_PATH, "%s/%s", src_dir, entry->d_name);
snprintf(dst_path, MAX_PATH, "%s/%s", dst_dir, src_path + strlen(base_src_dir) + 1);
```

Getting File Status

- Uses the **stat** function to get information about the file. If it fails, prints an error message and continues to the next entry.

```
struct stat st;
if (stat(src_path, &st) == -1) {
    perror("stat");
    continue;
}
```

Handling Directories

- If the entry is a directory, calls **create_directory** to create the corresponding directory in the destination.
- Recursively calls **copy_directory** to copy the contents of the subdirectory.

```
if (S_ISDIR(st.st_mode)) {
    create_directory(dst_path);
    copy_directory(src_path, dst_dir, base_src_dir);
```

Adding Files to the Buffer

- Locks the mutex to gain exclusive access to the buffer.
- If the buffer is full, waits on the **not_full** condition variable.
- If the **done** flag is set, unlocks the mutex and breaks out of the loop.
- Copies the source and destination paths into the buffer.
- Updates the buffer's **in** index and increments the count.
- Signals the **not_empty** condition variable to wake up worker threads.
- Unlocks the mutex.

```
    pthread_mutex_lock(&buffer.mutex);

    while (buffer.count == buffer.buffer_size && !buffer.done) {
        pthread_cond_wait(&buffer.not_full, &buffer.mutex);
    }

    if (buffer.done) {
        pthread_mutex_unlock(&buffer.mutex);
        break;
    }

    strcpy(buffer.buffer[buffer.in].src_path, src_path);
    strcpy(buffer.buffer[buffer.in].dst_path, dst_path);
    buffer.in = (buffer.in + 1) % buffer.buffer_size;
    buffer.count++;

    pthread_cond_signal(&buffer.not_empty);
    pthread_mutex_unlock(&buffer.mutex);
}
```

Closing the Directory

- Closes the directory when done.

```
closedir(dir);
```

Worker Thread

Infinite Loop

The worker thread runs in an infinite loop, continuously checking for tasks to perform until the **done** flag is set and the buffer is empty.

- This loop ensures that the worker thread keeps running and processing tasks as long as there are tasks available or until the manager signals that all tasks are done.

Locking the Mutex

- Locks the mutex to ensure exclusive access to the buffer. This prevents race conditions and ensures that only one thread can modify the buffer at a time.

```
pthread_mutex_lock(&buffer.mutex);
```

Waiting for Tasks

- If the buffer is empty (**buffer.count == 0**) and the **done** flag is not set (**!buffer.done**), the worker thread waits on the **not_empty** condition variable.

```
while (buffer.count == 0 && !buffer.done) {
    pthread_cond_wait(&buffer.not_empty, &buffer.mutex);
}
```

- **pthread_cond_wait** releases the mutex and puts the thread to sleep until the **not_empty** condition is signaled. When the thread wakes up, it reacquires the mutex.

Checking the Done Flag

- If the buffer is still empty (**buffer.count == 0**) but the **done** flag is set (**buffer.done**), it means the manager has finished adding tasks and there are no more tasks to process.

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

- The worker thread releases the mutex and exits the loop, effectively terminating the thread.

Retrieving a File Pair from the Buffer

- Retrieves a file pair from the buffer at the **out** index.
- Increments the **out** index cyclically using the modulo operator to wrap around the buffer size.

```
FilePair file_pair = buffer.buffer[buffer.out];
buffer.out = (buffer.out + 1) % buffer.buffer_size;
buffer.count--;
```

- Decrements the **count** to reflect that an item has been removed from the buffer.

Signaling the Not Full Condition

- Signals the **not_full** condition variable to wake up any threads waiting to add items to the buffer.

```
pthread_cond_signal(&buffer.not_full);
```

- This is important because it allows the manager thread (or any other producer threads) to add more items to the buffer once space is available.

Unlocking the Mutex

- Releases the mutex to allow other threads to access the buffer.

```
pthread_mutex_unlock(&buffer.mutex);
```

- This is critical to avoid deadlock and ensure that other threads can proceed with their operations.

Copying the File

- Calls the **copy_file** function to perform the actual file copying operation.

```
copy_file(file_pair.src_path, file_pair.dst_path);
```

- The **copy_file** function handles opening the source and destination files, reading from the source, and writing to the destination.

Copy File Function

Opening the Source File

- Attempts to open the source file in read-only mode using **open**.
- If the file cannot be opened (e.g., it does not exist or there are insufficient permissions), **open** returns **-1**.
- If an error occurs, **perror** is called to print an error message, and the function returns immediately without proceeding further.

```
int src_fd = open(src, O_RDONLY);
if (src_fd < 0) {
    perror("open src");
    return;
}
```

Opening the Destination File

- Attempts to open the destination file in write-only mode. If the file does not exist, it is created (**O_CREAT**), and if it exists, its contents are truncated (**O_TRUNC**).

```
int dst_fd = open(dst, O_WRONLY | O_CREAT | O_TRUNC, 0644);
if (dst_fd < 0) {
    perror("open dst");
    close(src_fd);
    return;
}
```

- The file permissions for the created file are set to **0644**, which means the owner can read and write, while others can only read.

- If the file cannot be opened or created, **open** returns **-1**.

- If an error occurs, **perror** is called to print an error message, the source file descriptor (**src_fd**) is closed, and the function returns.

Buffer Allocation

- Allocates a buffer of 8192 bytes for reading and writing data. This size is chosen to balance memory usage and performance.

Reading from Source and Writing to Destination

- Uses a loop to read data from the source file and write it to the destination file.
- **read** reads up to **sizeof(buffer)** bytes from the source file into the buffer. It returns the number of bytes actually read, or **-1** if an error occurs.

```
ssize_t bytes;
while ((bytes = read(src_fd, buffer, sizeof(buffer))) > 0) {
    if (write(dst_fd, buffer, bytes) != bytes) {
        perror("write");
        break;
    }

    num_bytes += bytes;
}
```

- The loop continues as long as **read** returns a positive number of bytes.
- **write** writes the data from the buffer to the destination file. It returns the number of bytes actually written, or **-1** if an error occurs.
- If **write** does not write the expected number of bytes, **perror** is called to print an error message, and the loop is broken.
- The number of bytes successfully copied is accumulated in **num_bytes**.

Closing the Files

- Closes both the source and destination file descriptors to release the resources.

```
close(src_fd);
close(dst_fd);
```

Signal Handling

Signal Handler Function

The **handle_signal** function is called when the program receives a signal. Here, it handles the **SIGINT** signal, which is sent when the user presses **Ctrl + C**.

Lock Mutex:

- Locks the mutex to ensure exclusive access to the buffer. This prevents race conditions while modifying shared data.

```
pthread_mutex_lock(&buffer.mutex);
```

Set Done Flag:

- Sets the **done** flag to 1. This flag indicates to all threads that they should stop processing and exit their loops.

```
buffer.done = 1;
```

Broadcast Conditions:

- Signals all threads waiting on the **not_empty** and **not_full** condition variables. This wakes up any threads that might be waiting for the buffer to become non-empty or non-full, allowing them to re-check the **done** flag and exit if necessary.

```
pthread_cond_broadcast(&buffer.not_empty);
pthread_cond_broadcast(&buffer.not_full);
```

Unlock Mutex:

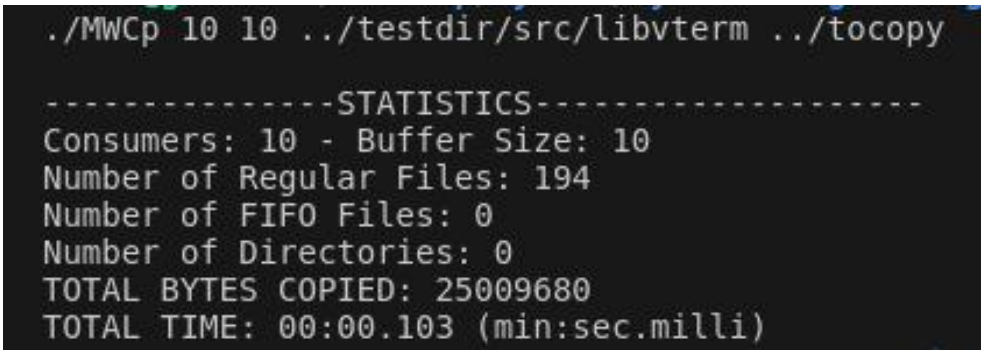
- Unlocks the mutex, allowing other threads to proceed and check the **done** flag.

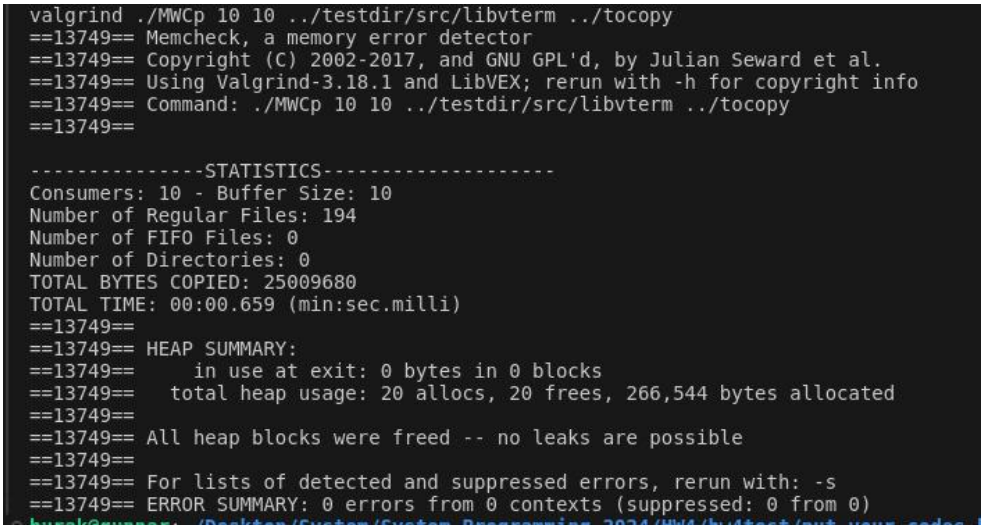
```
pthread_mutex_unlock(&buffer.mutex);
```

Test

Test1

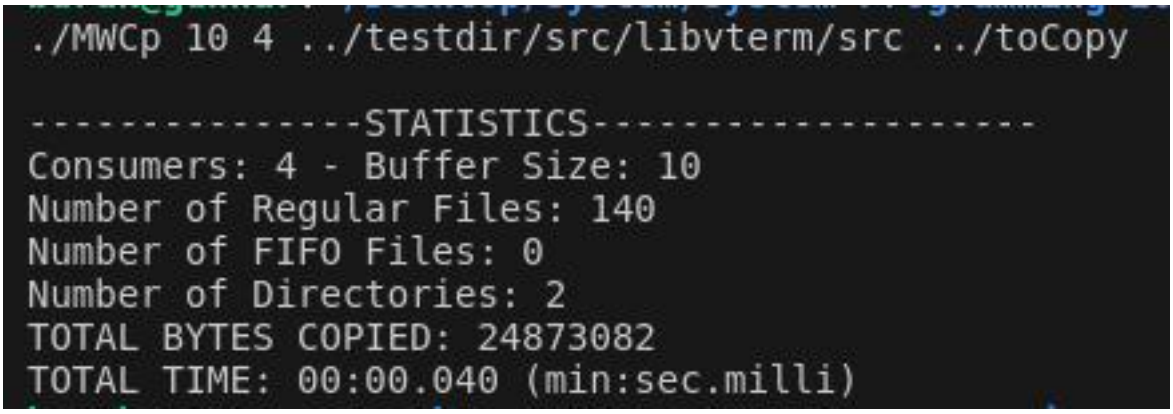
./MWCp 10 10 ../testdir/src/libvterm ../tocopy

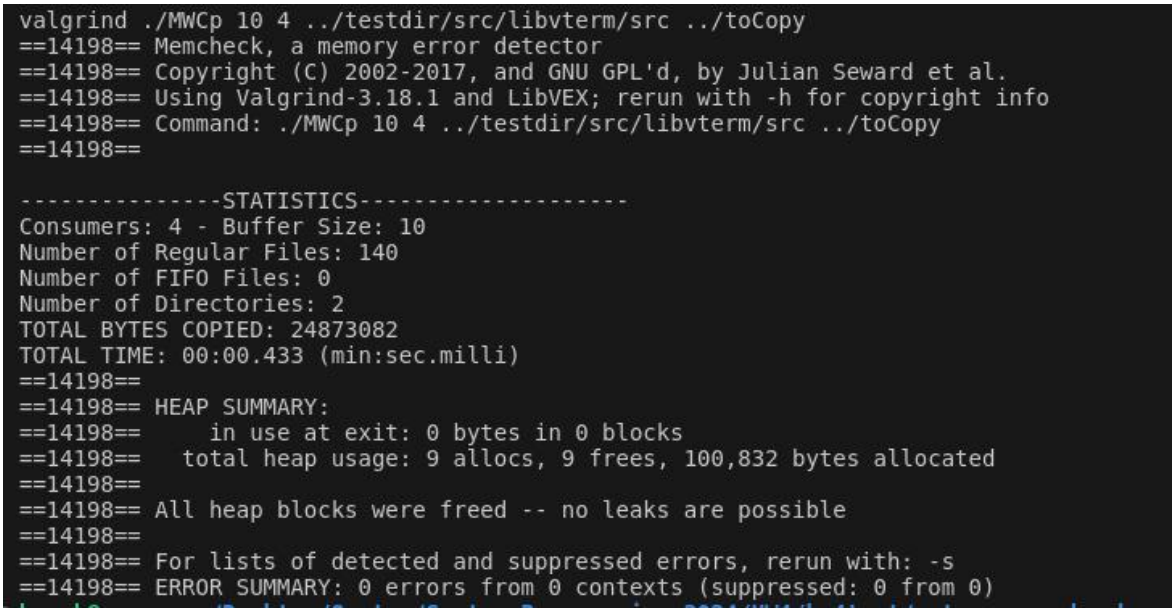




Test2

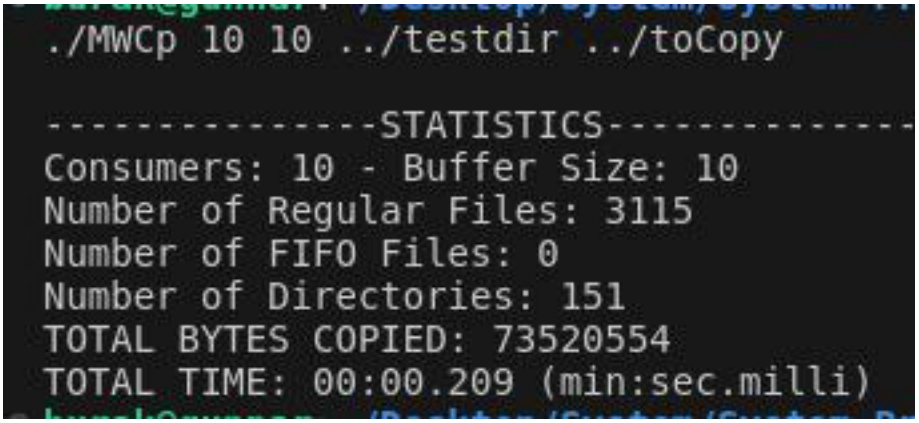
./MWCp 10 4 ../testdir/src/ libvterm/src ../toCopy





Test 3

./MWCp 10 10 ../testdir ../toCopy




```
valgrind ./MWCp 10 10 ../testdir ../toCopy
==14391== Memcheck, a memory error detector
==14391== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==14391== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==14391== Command: ./MWCp 10 10 ../testdir ../toCopy
==14391==

----------STATISTICS----------
Consumers: 10 - Buffer Size: 10
Number of Regular Files: 3116
Number of FIFO Files: 0
Number of Directories: 151
TOTAL BYTES COPIED: 73520554
TOTAL TIME: 00:01.584 (min:sec.milli)
==14391==
==14391== HEAP SUMMARY:
==14391==     in use at exit: 0 bytes in 0 blocks
==14391==   total heap usage: 164 allocs, 164 frees, 4,992,048 bytes allocated
==14391==
==14391== All heap blocks were freed -- no leaks are possible
==14391==
==14391== For lists of detected and suppressed errors, rerun with: -s
==14391== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

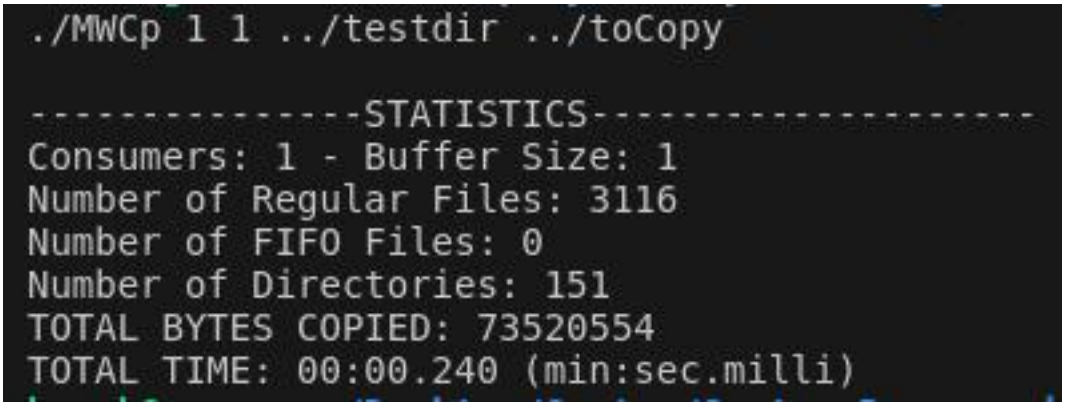
Test 3 with SIG_INT

```
valgrind ./MWCp 10 10 ../testdir ../toCopy
==14637== Memcheck, a memory error detector
==14637== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==14637== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==14637== Command: ./MWCp 10 10 ../testdir ../toCopy
==14637==
^CInterrupt signal received. Cleaning up...

---------------STATISTICS--------------------
Consumers: 10 - Buffer Size: 10
Number of Regular Files: 1149
Number of FIFO Files: 0
Number of Directories: 43
TOTAL BYTES COPIED: 9072146
TOTAL TIME: 00:00.650 (min:sec.milli)
==14637==
==14637== HEAP SUMMARY:
==14637==     in use at exit: 0 bytes in 0 blocks
==14637==   total heap usage: 56 allocs, 56 frees, 1,447,920 bytes allocated
==14637==
==14637== All heap blocks were freed -- no leaks are possible
==14637==
==14637== For lists of detected and suppressed errors, rerun with: -s
==14637== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Test 4

./MWCp 1 1 ../testdir ../toCopy




```
valgrind ./MWCp 1 1 ../testdir ../toCopy
==14893== Memcheck, a memory error detector
==14893== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==14893== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==14893== Command: ./MWCp 1 1 ../testdir ../toCopy
==14893==

----------STATISTICS----------
Consumers: 1 - Buffer Size: 1
Number of Regular Files: 3116
Number of FIFO Files: 0
Number of Directories: 151
TOTAL BYTES COPIED: 73520554
TOTAL TIME: 00:01.149 (min:sec.milli)
==14893==
==14893== HEAP SUMMARY:
==14893==     in use at exit: 0 bytes in 0 blocks
==14893==   total heap usage: 155 allocs, 155 frees, 4,989,600 bytes allocated
==14893==
==14893== All heap blocks were freed -- no leaks are possible
==14893==
==14893== For lists of detected and suppressed errors, rerun with: -s
==14893== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Test 5

```
./MWCp 1 10 ../testdir ../toCopy

----------------STATISTICS--------------------
Consumers: 10 - Buffer Size: 1
Number of Regular Files: 3116
Number of FIFO Files: 0
Number of Directories: 151
TOTAL BYTES COPIED: 73520554
TOTAL TIME: 00:00.238 (min:sec.milli)
```

```
valgrind ./MWCp 1 10 ../testdir ../toCopy
==14964== Memcheck, a memory error detector
==14964== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==14964== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==14964== Command: ./MWCp 1 10 ../testdir ../toCopy
==14964==

----------------STATISTICS--------------------
Consumers: 10 - Buffer Size: 1
Number of Regular Files: 3116
Number of FIFO Files: 0
Number of Directories: 151
TOTAL BYTES COPIED: 73520554
TOTAL TIME: 00:01.700 (min:sec.milli)
==14964==
==14964== HEAP SUMMARY:
==14964==     in use at exit: 0 bytes in 0 blocks
==14964==   total heap usage: 164 allocs, 164 frees, 4,992,048 bytes allocated
==14964==
==14964== All heap blocks were freed -- no leaks are possible
==14964==
==14964== For lists of detected and suppressed errors, rerun with: -s
==14964== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```