Alper Tavşanoğlu-210104004142-HW-5-Report

The purpose of this program is to perform parallel file and directory copying using threads and POSIX synchronization mechanisms. The program is designed to handle tasks such as copying files and directories to a new directory, with each task managed by a designated pool of worker threads in a synchronized computing environment. The system controls file and directory operations through thread synchronization, ensuring that each type of operation is handled by the appropriate worker thread. To achieve efficient task management, the program employs condition variables to signal the state of the buffer, enabling synchronized addition and removal of tasks by the manager and worker threads. Additionally, barriers are used to ensure that all worker threads reach a specific synchronization point before proceeding, ensuring that all threads complete a phase of processing before moving on to the next phase. The program runs until all specified files and directories have been successfully copied, demonstrating robust and synchronized parallel processing.

Changes Overview

Condition Variables

Usage

buffer_not_empty: Signals when the buffer is not empty, allowing worker threads to start processing.

buffer_not_full: Signals when the buffer is not full, allowing the manager thread to add more items.

Changes

Manager thread waits on **buffer_not_full** when the buffer is full.

Worker threads wait on **buffer_not_empty** when the buffer is empty.

Appropriate signaling is done to wake up the respective threads.

Barriers

Usage

A barrier is used to synchronize worker threads once they finish processing their tasks, ensuring the final statistics are printed only after all threads have completed their work.

Changes

Introduced **pthread_barrier_t** for synchronizing worker threads.

Worker threads wait at the barrier after completing their tasks.

Global Variables and Initialization

Added **pthread_cond_t buffer_not_empty** and **pthread_cond_t buffer_not_full** to the **SharedBuffer** struct.

Added **pthread_barrier_t** barrier for thread synchronization.

Initialized these synchronization primitives in the main function.

Manager Functionality

Modified the **add_task** function as;

Wait on **buffer_not_full** if the buffer is full.

Signal **buffer_not_empty** after adding a task to the buffer.

Worker Functionality

Modified the worker function as;

Wait on **buffer_not_empty** if the buffer is empty.

Signal **buffer_not_full** after removing a task from the buffer.

Wait at the barrier after completing their tasks to ensure all threads synchronize before concluding.

Global Initialization

```
// Shared buffer to hold file tasks and synchronization primitives
typedef struct{
        FileTask *tasks;
        int bufferSize;
        int numTasks;
        int nextTask;
        int done;
        pthread_mutex_t mutex;
        pthread_cond_t notEmpty;        // Condition variable for "not empty"
        pthread_cond_t notFull;         // Condition variable for "not full"
        pthread_barrier_t barrier;      // Barrier for synchronization
}SharedBuffer;
```

Add Task

```c
// Function to add a task to the shared buffer
void add_task(SharedBuffer *sharedBuffer, const char *srcPath, const char *destPath){
        pthread_mutex_lock(&sharedBuffer->mutex);                               // Lock the mutex
        // Expand the task buffer if necessary
        if(sharedBuffer->numTasks >= sharedBuffer->bufferSize){
                sharedBuffer->bufferSize *= 2;
                sharedBuffer->tasks = realloc(sharedBuffer->tasks, sharedBuffer->bufferSize * sizeof(FileTask));
                if(sharedBuffer->tasks == NULL){
                        fprintf(stderr, "Failed to expand task buffer\n");
                        exit(EXIT_FAILURE);
                }
        }
        // Add the new task to the buffer
        snprintf(sharedBuffer->tasks[sharedBuffer->numTasks].srcPath, MAX_PATH, "%s", srcPath);
        snprintf(sharedBuffer->tasks[sharedBuffer->numTasks].destPath, MAX_PATH, "%s", destPath);
        sharedBuffer->numTasks++;
        pthread_cond_signal(&sharedBuffer->notEmpty);                   // Signal that a new task is available
        pthread_mutex_unlock(&sharedBuffer->mutex);                     // Unlock the mutex
}
```

worker

```c
// Worker thread function to process file tasks from the buffer
void *worker(void *arg){
        SharedBuffer *sharedBuffer = (SharedBuffer *)arg;
        while(!terminate){                                              // Loop until termination flag is set
                pthread_mutex_lock(&sharedBuffer->mutex);
                // Wait for tasks to be available or for the done flag to be set
                while(sharedBuffer->nextTask >= sharedBuffer->numTasks && !sharedBuffer->done && !terminate){
                        pthread_cond_wait(&sharedBuffer->notEmpty, &sharedBuffer->mutex);       // Wait for condition variable
                }
                if(terminate || (sharedBuffer->nextTask >= sharedBuffer->numTasks && sharedBuffer->done)){
                        pthread_mutex_unlock(&sharedBuffer->mutex);
                        break;
                }
                // Get the next task from the buffer
                FileTask task = sharedBuffer->tasks[sharedBuffer->nextTask++];
                pthread_cond_signal(&sharedBuffer->notFull);            // Signal that buffer is not full now
                pthread_mutex_unlock(&sharedBuffer->mutex);
```

in main

```c
    SharedBuffer sharedBuffer;
    sharedBuffer.bufferSize = userBufferSize;
    sharedBuffer.tasks = malloc(sharedBuffer.bufferSize * sizeof(FileTask));
    sharedBuffer.numTasks = 0;
    sharedBuffer.nextTask = 0;
    sharedBuffer.done = 0;
    pthread_mutex_init(&sharedBuffer.mutex, NULL);
    pthread_cond_init(&sharedBuffer.notEmpty, NULL);
    pthread_cond_init(&sharedBuffer.notFull, NULL);
    pthread_barrier_init(&sharedBuffer.barrier, NULL, numWorkers + 1);      // Include main thread in the barrier
```

System Design

Main Program

The program accepts buffer size, number of workers, and source/destination directory paths as input arguments. It checks for the correct number of arguments and prints a usage message if they are incorrect. It initializes and manages worker threads, ensuring efficient task distribution and synchronization. All mutexes, condition variables, and barriers are destroyed correctly.

Condition variables are used to manage worker cycles, ensuring proper synchronization without busy waiting. Barriers are implemented to ensure that all worker threads reach a specific synchronization point before proceeding, allowing all threads to complete a phase of processing before moving to the next phase.

The program tracks and reports the time taken to copy the entire directory content, along with detailed statistics about the types and numbers of files copied.

Signal handling is implemented to allow the program to terminate gracefully in response to interruptions like SIGINT or SIGTERM. A custom signal handler sets a termination flag, which is checked by threads to conclude their operations safely. The program records and outputs detailed statistics about the operation, fulfilling a key requirement.

Manager

In the program, the manager is implemented within the main function. It reads from the source directory and prepares tasks for workers by creating appropriate destination files, handling truncation if files already exist. The manager provides robust error reporting if file operations fail and manages file descriptors securely. It controls a shared buffer, ensuring that workers can access tasks as they are ready without conflicts or resource contention.

The SharedBuffer is central to synchronization, equipped with mutexes, condition variables, and barriers. It stores file tasks, tracks the number of tasks, and signals task availability to workers. This buffer is critical for managing the flow of data between the manager and workers, ensuring that no two workers process the same file simultaneously.

The manager uses condition variables to manage the state of the buffer, signaling when tasks are added and when space is available for more tasks. Barriers are used to ensure that all worker threads reach a specific synchronization point before proceeding, which allows the manager to signal the completion of task preparation and enable the graceful termination of worker threads.

Worker

Workers pull tasks from the shared buffer, handling file copying from the source to the destination. This involves opening source files, reading content, and writing to destination files while managing file-specific actions such as truncation and error handling. Each worker is responsible for managing its file descriptors and allocated resources, ensuring that all resources are freed and closed appropriately after task completion.

Workers utilize mutexes, condition variables, and barriers from the shared buffer to synchronize their access to the buffer, ensuring efficient and conflict-free task processing. They wait on condition variables when the buffer is empty and signal the manager when they remove tasks, maintaining a smooth flow of tasks.

Barriers ensure that all workers reach a specific synchronization point before proceeding, which helps in managing the overall task flow. Workers handle their termination based on the shared termination flag set by the signal handler, allowing them to exit gracefully when the program receives a termination signal.

Basic Pseudocodes

```
Initialize shared buffer and synchronization primitives
Parse command line arguments
Set up signal handler for CTRL+C

Start timing
Handle source and destination paths to load tasks into the buffer

Create worker threads

Wait for all tasks to be processed
Notify workers of completion
Join worker threads

End timing

Print statistics

Clean up resources
```

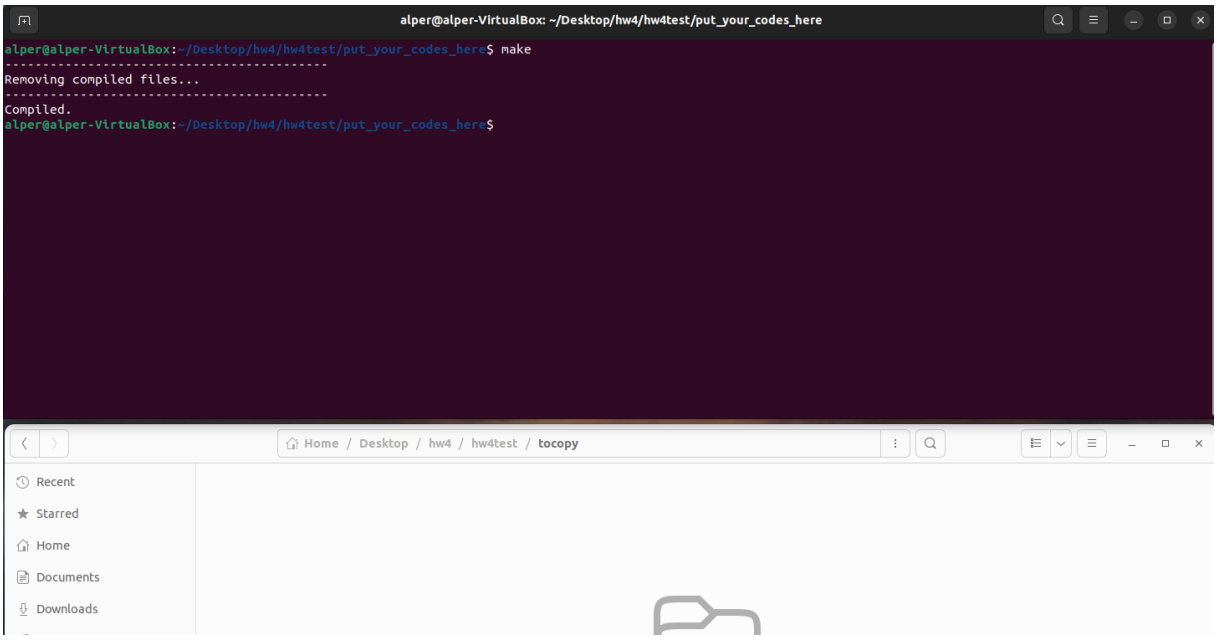
```
While not terminated:
    Lock mutex
    Wait for tasks to be available in buffer or done flag to be set
    Unlock mutex if no tasks are available and done flag is set

    Lock mutex
    Read task from buffer
    Unlock mutex

    Open source file
    Open destination file with truncation

    Copy data from source file to destination file

    Close source and destination files

    Lock mutex
    Update statistics
    Unlock mutex

    Print completion status
```

Compile and Tests

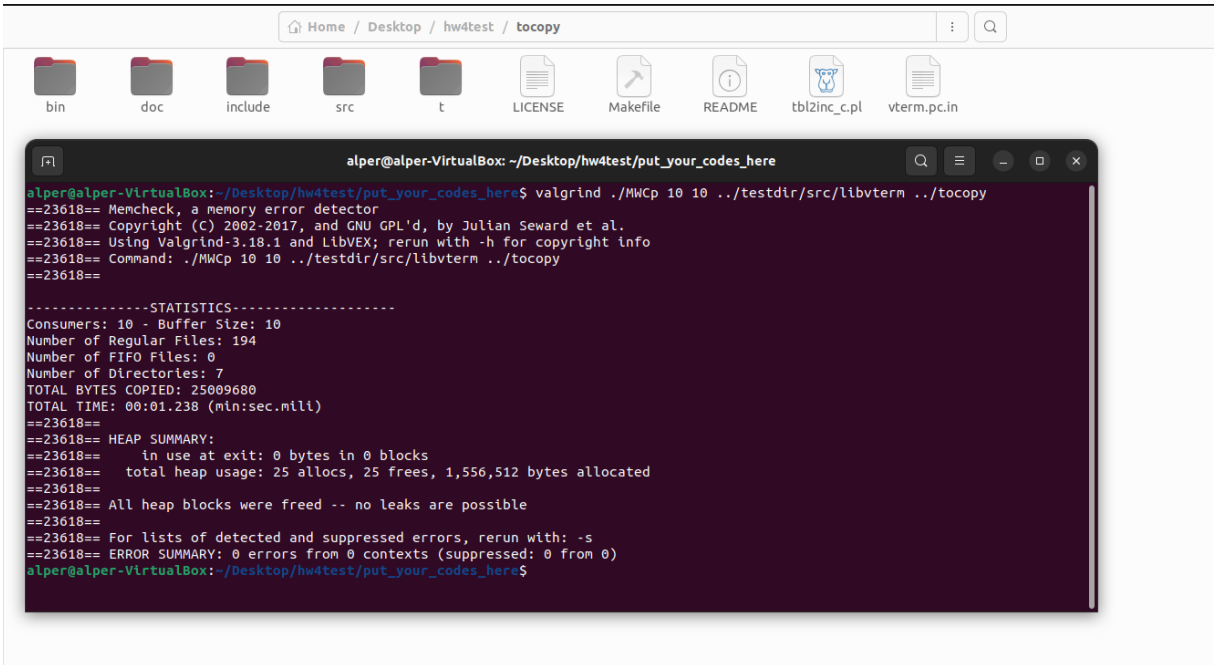
For compile the code we can use Makefile. With **make** command program will compiled. Makfile also include **makeclean** command.

Test1: valgrind ./MWCp 10 10 ../testdir/src/libvterm ../tocopy, #memory leak checking, buffer size=10, number of workers=10, sourceFile, destinationFile

First Compile and Before Run Test1

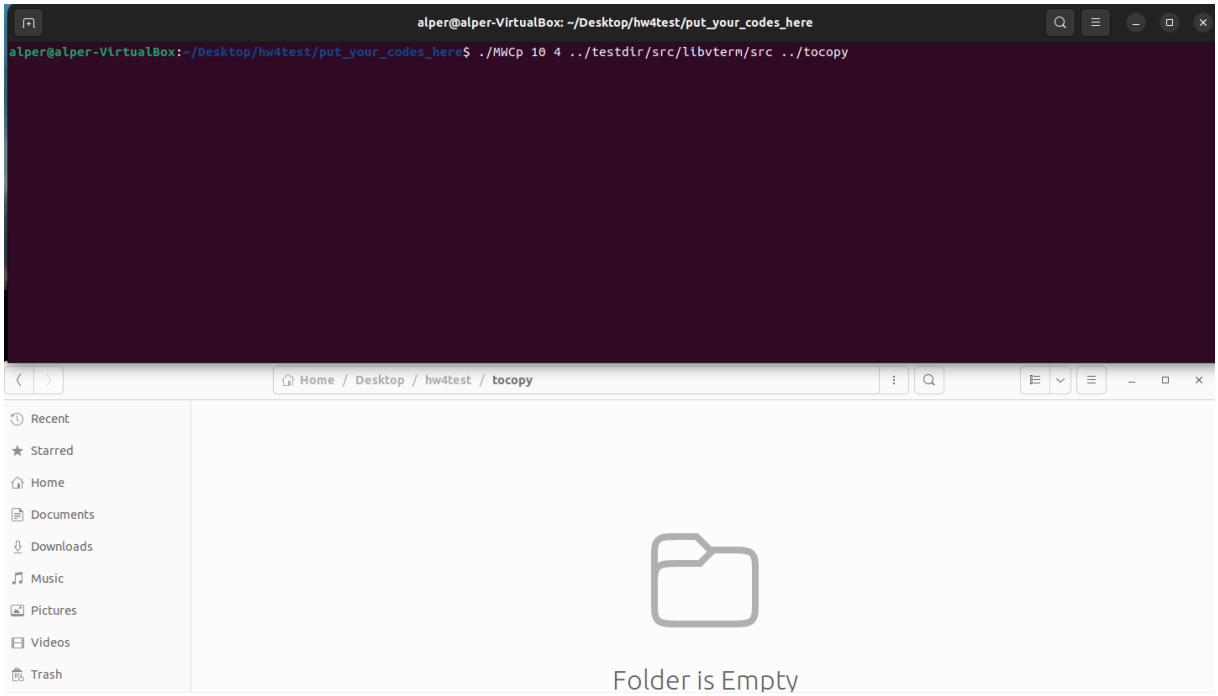


Then Run Test1

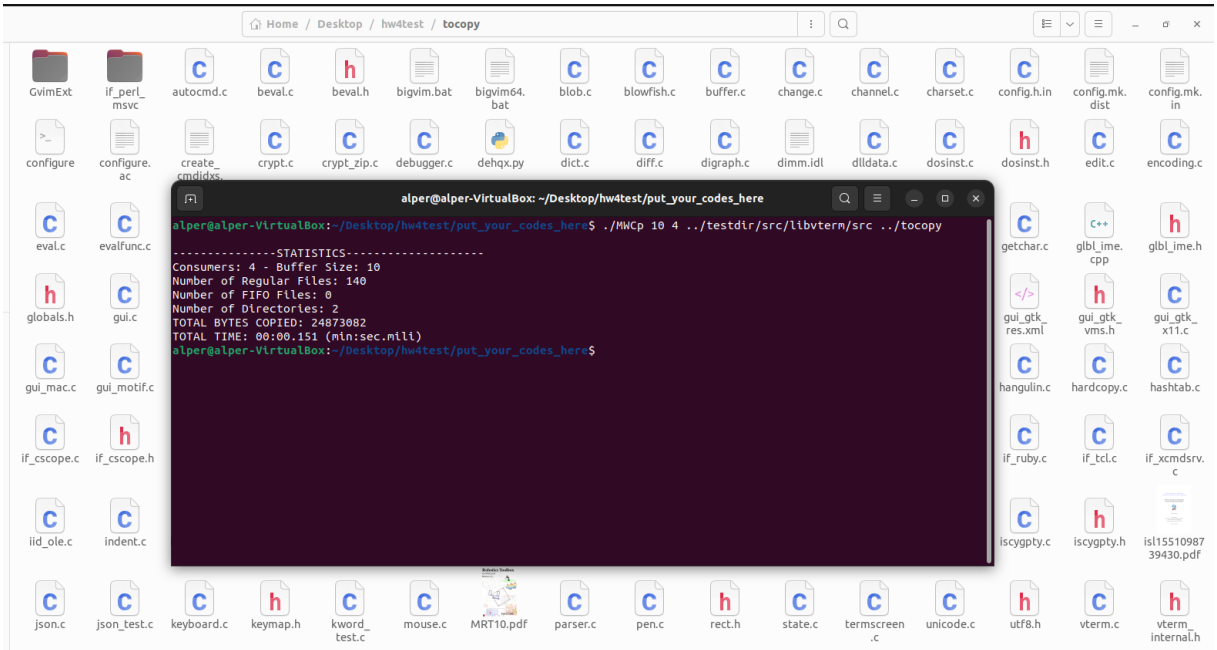


Test2: ./MWCp 10 4 ../testdir/src/ libvterm/src ../tocopy #buffer size=10, number of workers=4, sourceFile, destinationFile

Before Run Test2

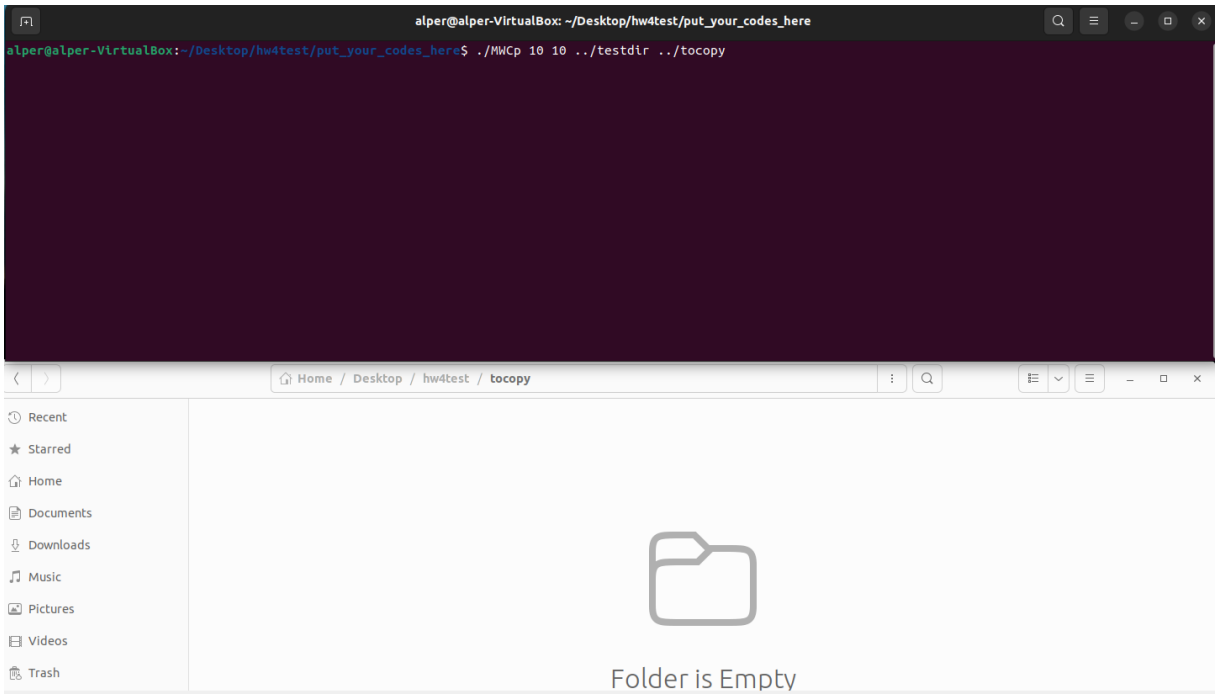


After Run Test2



Test3: ./MWCp 10 10 ../testdir ../tocopy #buffer size=10, number of workers=10, sourceFile, destinationFile

Before Run Test3



Run Test3