Gebze Tecnhical University

CSE344 System Programming

-

Homework #5 Report

Abdulsamet Aslan

200104004098

## Problem overview

The problem addressed in this project is the efficient copying of files and sub-directories within a large directory tree using a utility called "pCp." Existing utilities may struggle to handle the resource requirements when working with such large directory trees. To overcome this limitation, the implementation of pCp aims to achieve parallel processing by creating a new thread for each file and sub-directory copy operation.

## Problem Solving Approach

*Checking the given paths*

Main function checks the source and destination paths to prevent to recursion infinite copying into source itself.

*Declaration of the Buffer*

I used a queue implementation to create a buffer that stores the file description and filenames.

*Creating Threads*

After taking commands from user, a producer thread and consumer threads which are taken from command line are created. While producer thread takes directory names as parameter, consumer threads doesn't take any parameter.

*Mutexes and Condition Variables*

I created two different mutexes for synchronization between threads. First one is created to establish the write-read synchronizations. Second one is used to protect the critical section which the producer and the multiple consumers are writing to the standard output.

Also, I created another mutex to prevent to exceed the limit of the opened file descriptors. When it reaches the limit, it just waits for another file descriptor to close.

*Producer Thread*

Firstly, It checks whether destination file exists. If it exists, creates a directory that shares same name with source directory. Then producer thread calls a recursive function to copy subdirectories. It finds the files and directories in the subdirectory. If there is a directory, it calls itself recursively. Otherwise, it opens the source file to read and destination file to write. Finally, it locks the required mutex and push the file descriptors and filename to queue. Then it unlocks the mutex and signal to 'full' condition variable to inform consumer threads.

*Consumer Threads*

Consumer Threads locks the mutex and pop an element from the buffer, signals to 'empty' condition variable to inform producer thread and unlocks the mutex. After that It reads from source file descriptor and writes to destination file

descriptor as chunks. If a signal occurs when this process is executing, the consumer threads finish their work with the last chunk at that moment and breaks the loop to terminate.

*Handle signals*

Program handles the SIGINT AND SIGTSTP signals by assigning done variable and signal_received variables. These variables informs the threads to be terminated. Threads check these variables and finish them last operations and terminate.

*Terminating Threads*

End of the main loop, program waits for the thread to terminate using *phtread_join.* Furthermore, it destroys all threads and condition variables.

## Tests

(78 MB directory copied for testing)

1000 buffer size – 1 thread -> 37 seconds

1000 buffer size – 100 thread -> 42 seconds

1 buffer size – 1 thread -> 42 seconds

1 buffer size – 100 thread -> 35 seconds

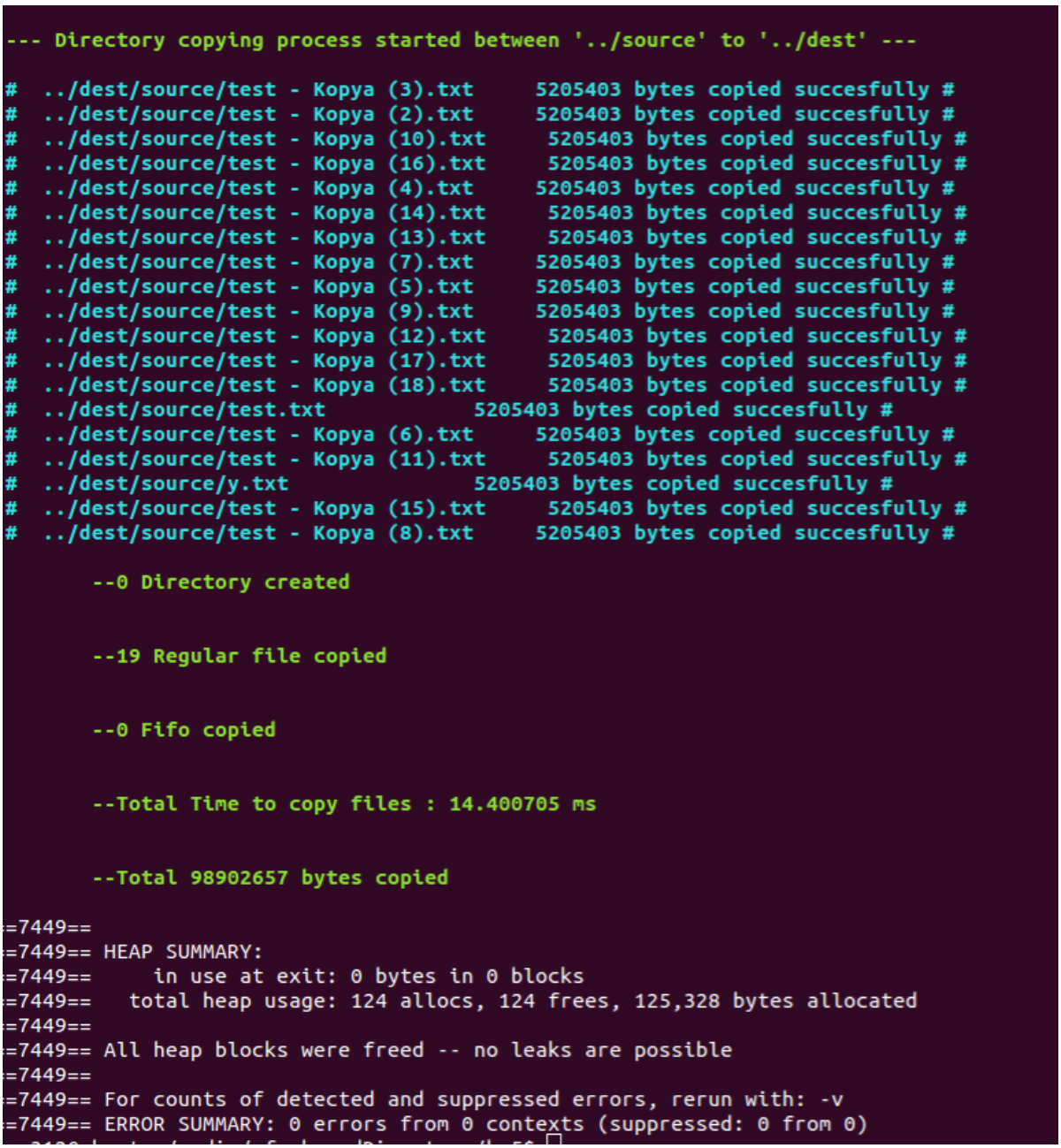
1 buffer size – 1000 thread -> 36 seconds

100 buffer size – 100 thread -> 35 seconds

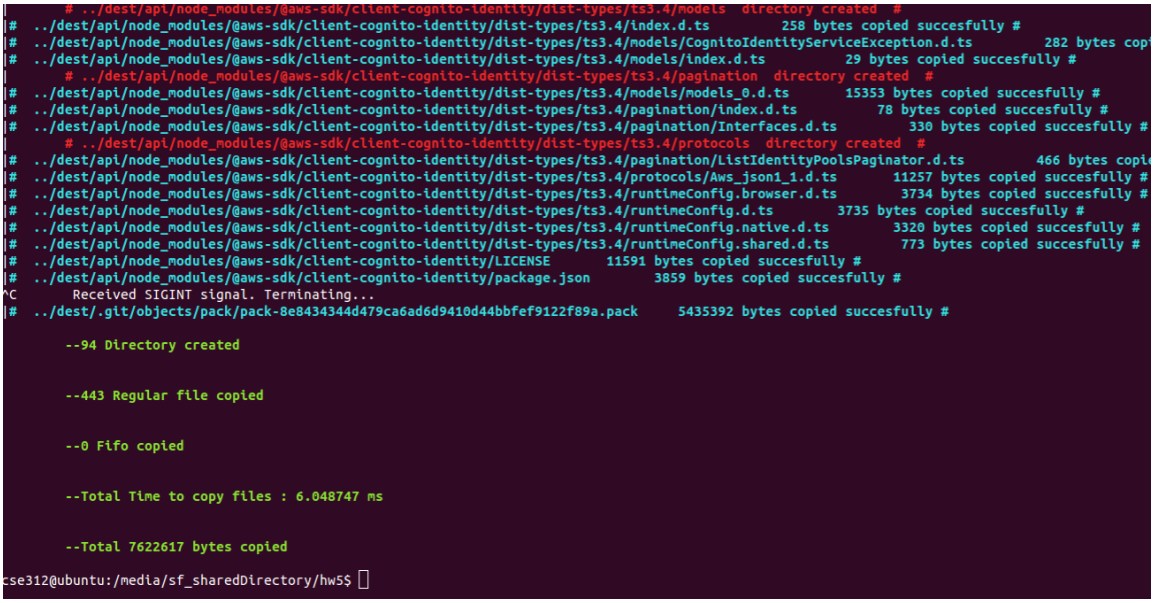
## Result

According to my tests, When the number of threads is close to the buffer size, I can achieve the best results. The optimal performance was obtained with a buffer size of 100 and 100 threads. However, in some cases, having a large number of threads can lead to time loss when joining them and creating them.

## Memory leak check



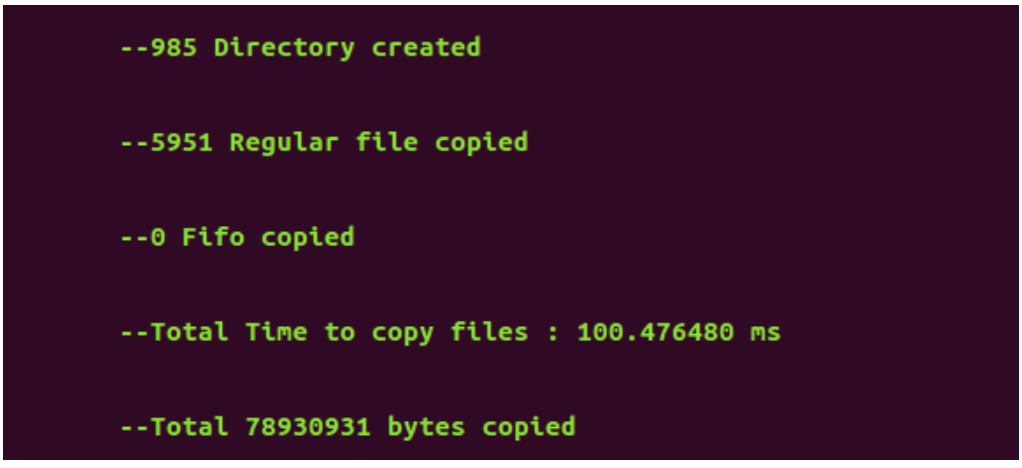
## Signal handling Check




*What happens when program exceed the per-process limit on the number of open of file descriptors?*

My program checks the number of open file descriptors with counter. When it reaches the limit, producer thread starts to wait in a while loop using condition variable. Once a consumer finishes its job with a file descriptor, it sends a signal to producer and producer continues to execute.

## ./hw5 10000000 1 ../source2 ../dest (6000 file copied)



# Compiling and Running

### *Compiling*

```
make
```

### *Running*

```
./hw5  <buffer_size>  <num_consumers>  <source_file>  <dest_file>
```

### *Valgrind*

```
Valgrind  ./hw5  <buffer_size>  <num_consumers>  <source_file>
<dest_file>
```

### *Clean the unnecessary files*

```
make clean
```