CSE 344

System Programming

5th Assignment

Report

Author

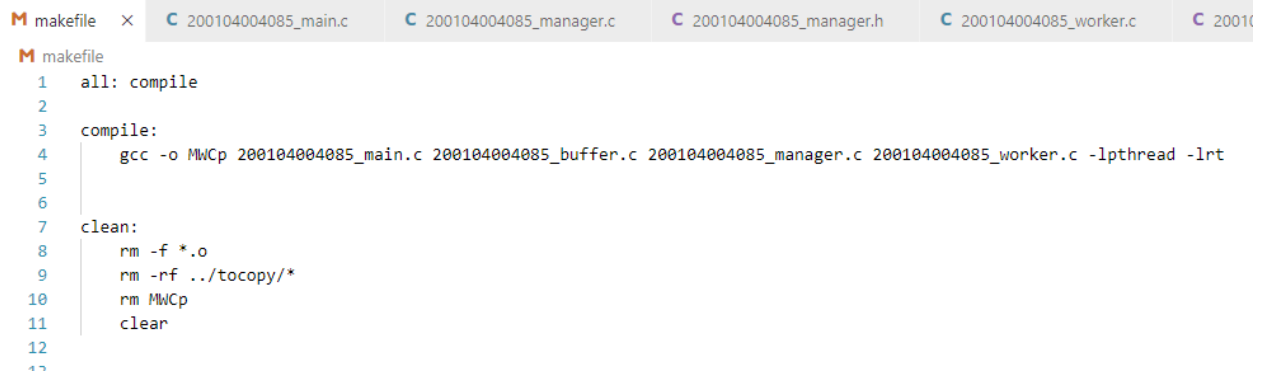Mert Emir ŞEKER

200104004085

Date

29.05.2024

Table Of Contents

1. Makefile:

2. Makefile Commands:

All: Compiles the code.

Compile: Compiles the code.

Clean: Clears terminal and deletes the tocopy's inside.

3. How to run the code?

To run the program, you can write **make all** or just **make** in terminal and then **./MWCp #buffer_size #ofworkers ../source_file_directory ../destination_file_directory**.

4. Key changes from HW4:

In HW5 we are asked to implement using conditional variables and barriers. In HW4 I used conditional variables. In HW5 I made an improvement by adding barriers to the code I used in HW4. Here are the places I changed:

buffer.h: I added **pthread_barrier_t barrier** to the **buffer_str** structure.

buffer.c: I initialized the barrier with **pthread_barrier_init** in the **buffer_init** function. I destroyed the barrier with **pthread_barrier_destroy** in the **buffer_destroy** function.

worker.c: I added **pthread_barrier_wait(&buffer->barrier);** to synchronize all worker threads at the end of the copying process.

manager.c: I added **pthread_barrier_wait(&buffer->barrier);** to ensure all threads reach this point before proceeding at the end of the manager function.

buffer.c: I modified the **buffer_destroy** function to check if the file descriptors are valid before closing them:

if (buffer->fds[idx].src_fd != -1)

 {

   close(buffer->fds[idx].src_fd);

}

if (buffer->fds[idx].dst_fd != -1)

{

   close(buffer->fds[idx].dst_fd);

}

5. <u>Code Explanation and pseudocodes:</u>

5.1. Buffer.c and Buffer.h: The buffer structure for handling file descriptors and the corresponding synchronization techniques required for worker threads to copy files in parallel is defined and put into practice by the buffer.h and buffer.c files. The buffer_init function initializes the buffer, setting up mutexes, condition variables, indices, and a barrier for thread coordination, among other synchronization primitives. By locking the mutex, ending file descriptors, releasing memory that has been allotted, and eliminating synchronization primitives, buffer_destroy frees up resources. By employing a mutex lock and condition variables to manage full buffer states and indicate when new items are added, the buffer_add method reliably adds file descriptors to the buffer. Similarly, buffer_remove uses condition variables to wait and indicate changes when extracting file descriptors, guaranteeing thread safety and synchronization, particularly when the buffer is empty. The done flag is set by the buffer_set_done function, which also notifies all waiting threads of the completion by broadcasting this status. This approach makes it possible to handle file descriptors effectively and synchronously, which is essential for the directory copying utility's simultaneous copying operation.

Pseudocode:

```
// Initialize the buffer
function buffer_init(buffer, size)
    set buffer size
    set buffer count to 0
    set buffer done flag to 0
    set buffer input index to 0
    set buffer output index to 0
    allocate memory for file descriptors
    if memory allocation fails
        print error and exit
    initialize mutex
    initialize not full condition variable
    initialize not empty condition variable
    initialize barrier with num_workers + 1

// Destroy the buffer and free resources
function buffer_destroy(buffer)
    lock mutex
    for each file descriptor in the buffer
        if source file descriptor is valid
            close source file descriptor
        if destination file descriptor is valid
            close destination file descriptor
    unlock mutex
    free file descriptor memory
    destroy mutex
    destroy not full condition variable
    destroy not empty condition variable
```

destroy barrier

// Add a file descriptor to the buffer

function buffer_add(buffer, fd)

    lock mutex

    while buffer is full and not done

        wait for not full condition

    if done

        unlock mutex and return

    add file descriptor to buffer at input index

    update input index

    increment buffer count

    signal not empty condition

    unlock mutex

// Remove a file descriptor from the buffer

function buffer_remove(buffer)

    lock mutex

    while buffer is empty and not done

        wait for not empty condition

    if buffer is empty and done

        unlock mutex and return invalid file descriptor

    get file descriptor from buffer at output index

    update output index

    decrement buffer count

    signal not full condition

    unlock mutex

    return file descriptor

// Set the done flag of the buffer

function buffer_set_done(buffer)

    lock mutex

    set done flag

    broadcast not empty condition

    broadcast not full condition

    unlock mutex

5.2. **Manager.c and Manager.h:** The directory copying utility's manager thread's functionality is defined and implemented by the manager.h and manager.c files. The buffer and source and destination directory locations, as well as function prototypes for controlling the directory copying operation, are stored in the manager_args structure provided in manager.h. The implementation of these functions may be found in the manager.c file. create_directory_if_not_exists checks to see if a given directory already exists and, if not, creates it. The function ensure_parent_directories_exist verifies the existence of each parent directory for a specified file path. copy_file takes care of adding file descriptors to the buffer, updating pertinent counters, and copying a file or establishing a FIFO. Process_directory copies files and directories in a directory while updating counters in a recursive manner. By identifying the kind of source path—directory, FIFO, or ordinary file—and executing the relevant methods to handle them, the manager function starts the process by making sure directories are established and files are copied. When copying is finished, it marks the buffer as done and uses a barrier to wait for all worker threads to synchronize. This approach controls tasks and worker-manager thread interactions to enable synchronized and efficient copying of directories and files.

Pseudocode:

// Function to create a destination directory if it does not exist

function create_directory_if_not_exists(dir_path)

   if directory does not exist

     create directory

     if creation fails

       print error and exit

// Function to ensure the parent directories exist for the given file path

function ensure_parent_directories_exist(file_path)

   duplicate file path

   find the last slash in the path

```
    if last slash exists

        terminate string at the last slash

        create directory if it does not exist

    free duplicated string


// Function to copy a file from source to destination

function copy_file(src_path, dst_path, buffer, mode)

    ensure parent directories exist

    if file is FIFO

        create FIFO file

        if creation fails

            print error

            return

        increment FIFO file counter

    else // Regular file

        open source file

        if opening fails

            print error

            return

        open destination file

        if opening fails

            print error and close source file descriptor

            return

        create file descriptor structure

        add file descriptor to buffer

        increment regular file counter


// Function to process a directory

function process_directory(src_dir, dst_dir, buffer)
```

ensure destination directory exists

open source directory

if opening fails

    print error and set buffer as done

    return

while reading directory entries and not done

    if entry is . or ..

        continue

    construct source path

    construct destination path

    get file status

    if getting status fails

        print error and continue

    if entry is a directory

        recursively process directory and increment directory counter

    else if entry is a FIFO

        copy FIFO file

    else if entry is a regular file

        copy regular file

close source directory

// Manager function to handle copying process

function manager(arg)

    cast argument to manager_args pointer

    get buffer from arguments

    get source and destination paths from arguments

    get status of source path

    if getting status fails

        print error and set buffer as done

return NULL

if source path is a directory

create root destination directory

ensure root destination directory exists

process source directory

else if source path is a FIFO

construct destination file path

copy FIFO file

else if source path is a regular file

construct destination file path

copy regular file

else

print error for unsupported file type and set buffer as done

return NULL

set buffer as done

wait for all worker threads to reach this point

return NULL

5.3 **Worker.c and Worker.h:** The directory copying utility's worker thread functionality is defined in the worker.h and worker.c files. The shared buffer for file descriptors is represented by a buffer_str pointer, which is converted to the parameter of the worker function in worker.c. After allocating memory for a buffer to read from and write to files, it starts a loop that uses buffer_remove to retrieve file descriptors until the global done variable indicates that the loop should end. It transfers data from the source file to the destination file while properly managing mistakes and interruptions. File descriptors are closed and a message is printed after copying. Using a mutex to update the global total_bytes_copied variable securely, the worker keeps track of the total number of bytes copied. To ensure effective and synchronized file copying across many threads, the worker thread then waits at a barrier for synchronization with other threads before clearing the allocated buffer and quitting.

Pseudocode:

```
// Worker thread function
function worker(arg)
    cast arg to buffer_str pointer
    allocate memory for buffer
    if memory allocation fails
        print error and return


    initialize total bytes copied by this thread to 0


    while not done
        get file descriptors from buffer
        if file descriptors are invalid
            break


        while read data from source file into buffer
            if read fails and is not interrupted by signal
                print error
                close file descriptors
                free buffer
                return


        write data from buffer to destination file
        while data remains to be written
            if write fails and is not interrupted by signal
                print error
                close file descriptors
                free buffer
                return
```

update write pointer and bytes remaining

update total bytes copied by this thread

if read fails and is not interrupted by signal

print error

close file descriptors

print copy completion message

lock mutex

update global total bytes copied

unlock mutex

wait for all threads to finish using barrier

free buffer

return

5.4 Main.c: The directory copying utility's main.c file handles the main logic, initializing global variables and managing command-line options for source, destination, buffer size, and worker count. It initializes the buffer, configures SIGINT signal handling for graceful termination, and starts worker threads and a manager thread for handling file copying and directory processing, respectively. To guarantee effective, parallel file copying, execution time is monitored and threads are synchronized with the use of mutexes and barriers. A thorough and synchronized method of parallel directory copying is provided by the destruction of the buffer, memory release, and printing of statistics, which include the total number of files transferred, total bytes copied, and elapsed time, once all threads have finished their duties.

```
// Function to print usage message
function print_usage(prog_name)
    print "Usage: prog_name <buffer_size> <num_workers> <src_dir> <dst_dir>"


// Signal handler for SIGINT
function handle_signal(sig)
    if sig is SIGINT
        print "Ctrl-C signal received. Copying stopped."
        set interrupted and done flags to 1
        set buffer as done


// Function to print statistics
function print_statistics(num_workers, buffer_size, elapsed)
    calculate elapsed time in minutes, seconds, and milliseconds
    print statistics including number of files, directories, total bytes copied, and total time


// Main function
function main(argc, argv)
    if argc is not 5
        print_usage(argv[0])
        exit with failure


    parse buffer_size, num_workers, src_dir, dst_dir from argv
    if buffer_size or num_workers is less than or equal to 0
        print_usage(argv[0])
        exit with failure


    print welcome message
```

set up signal handling for SIGINT

create destination directory if it does not exist

start measuring time

initialize buffer with buffer_size

print "Files are copying..."

create and start manager thread with src_dir, dst_dir, and buffer

allocate memory for worker threads
if allocation fails
    print error and exit with failure

create and start worker threads with buffer

wait for manager thread to finish

wait for all worker threads to finish

destroy buffer and free memory
stop measuring time and calculate elapsed time

print statistics with num_workers, buffer_size, and elapsed time
print "Files copy operation completed successfully. Goodbye!"
return 0

6. Example Outputs:

6.1. Test 1 (in pdf):

6.2. Test 2 (in pdf):

6.3. Test 3 (in pdf):

6.4. ./MWCp 10 10 ../testdir ../tocopy:

6.5. valgrind ./MWCp 1024 10 ../testdir ../tocopy:

6.6. ./MWCp 1024 10 ../testdir ../tocopy:

6.7. valgrind ./MWCp 10 4 ../testdir/src/libvterm/src ../newfile:

6.8. valgrind ./MWCp 10 10 ../testdir ../toCopy:

6.9. valgrind ./MWCp 10 10 ../testdir ../tocopy (with Ctrl-C signal):