Gebze Technical University

Computer Engineering Faculty

Homework 5 Report

System Programming

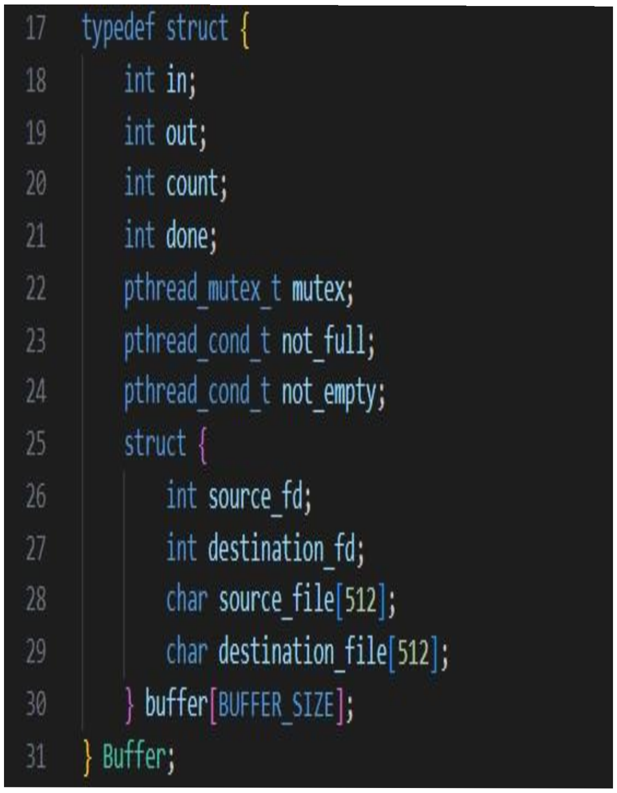
Student: Burak Ersoy

No: 1901042703

What is the my program?

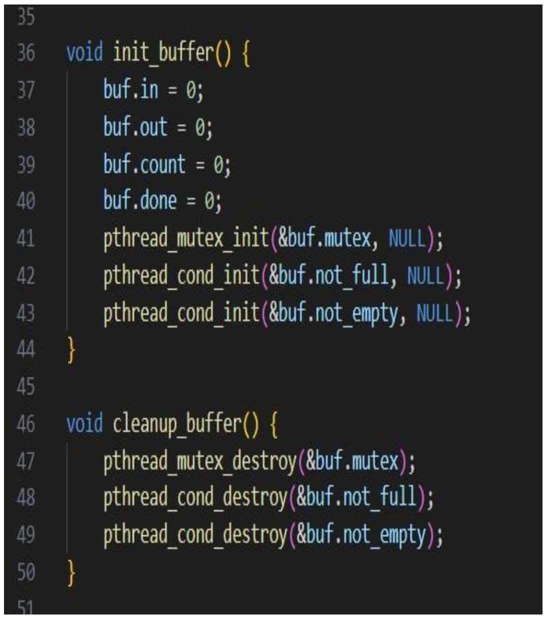
This program demonstrates a multi-threaded approach for parallel file copying, where the producer thread generates file descriptor pairs by reading files from a source directory, and consumer threads consume these pairs by copying the file content from the source to the destination directory.

Function Explanation
1-



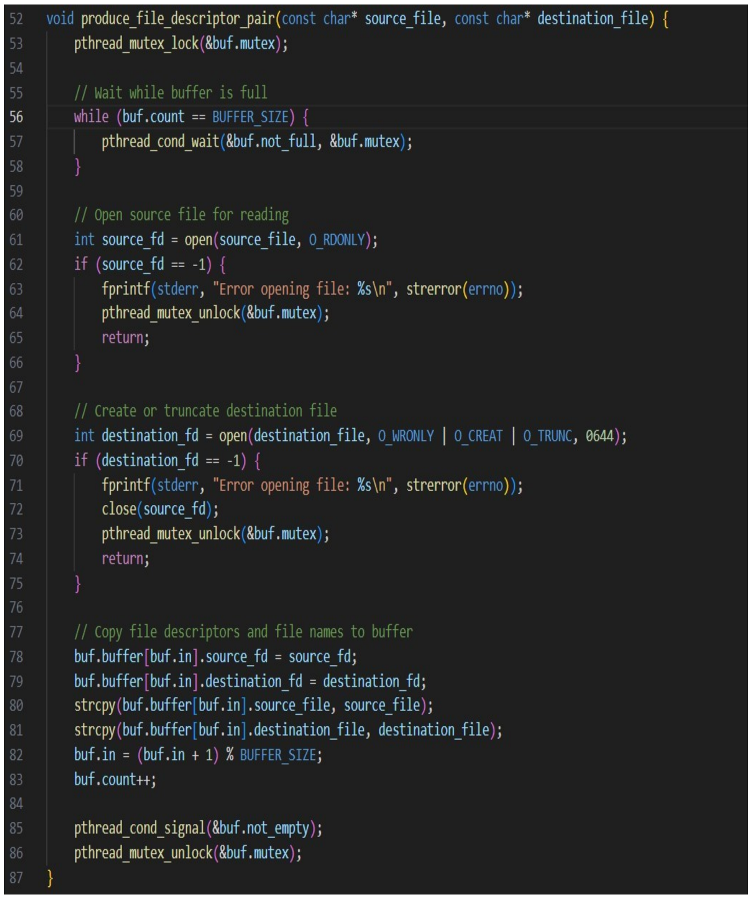
```c
typedef struct {
    int in;
    int out;
    int count;
    int done;
    pthread_mutex_t mutex;
    pthread_cond_t not_full;
    pthread_cond_t not_empty;
    struct {
        int source_fd;
        int destination_fd;
        char source_file[512];
        char destination_file[512];
    } buffer[BUFFER_SIZE];
} Buffer;
```

This Buffer structure is used as a shared data structure between producer and consumer threads. It provides a circular buffer with synchronization mechanisms to safely produce and consume file descriptor pairs for copying files. The in, out, count, and done variables manage the buffer's state, the mutex ensures thread safety, and the not_full and not_empty condition variables allow threads to wait and be notified when the buffer is ready for production or consumption. The buffer array holds the file descriptor pairs and associated file names.



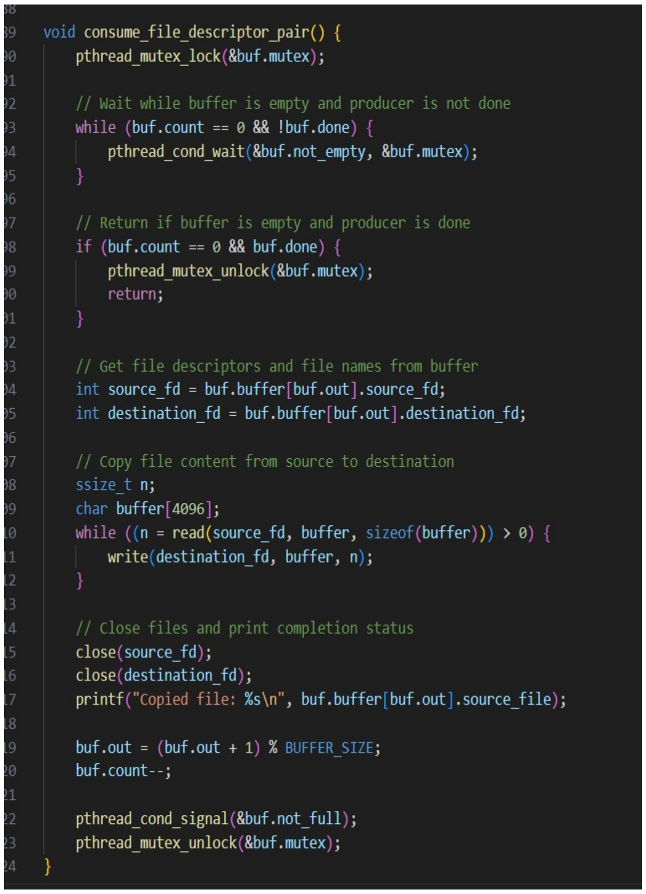
```c
void init_buffer() {
    buf.in = 0;
    buf.out = 0;
    buf.count = 0;
    buf.done = 0;
    pthread_mutex_init(&buf.mutex, NULL);
    pthread_cond_init(&buf.not_full, NULL);
    pthread_cond_init(&buf.not_empty, NULL);
}

void cleanup_buffer() {
    pthread_mutex_destroy(&buf.mutex);
    pthread_cond_destroy(&buf.not_full);
    pthread_cond_destroy(&buf.not_empty);
}
```

2-
init_buffer() initializes the Buffer structure and its associated synchronization mechanisms (mutex and condition variables), while cleanup_buffer() cleans up and destroys those synchronization mechanisms. These functions ensure proper initialization and cleanup of the shared data structure, making it ready for use and releasing any acquired resources when they are no longer needed.



```c
void produce_file_descriptor_pair(const char* source_file, const char* destination_file) {
    pthread_mutex_lock(&buf.mutex);

    // Wait while buffer is full
    while (buf.count == BUFFER_SIZE) {
        pthread_cond_wait(&buf.not_full, &buf.mutex);
    }

    // Open source file for reading
    int source_fd = open(source_file, O_RDONLY);
    if (source_fd == -1) {
        fprintf(stderr, "Error opening file: %s\n", strerror(errno));
        pthread_mutex_unlock(&buf.mutex);
        return;
    }

    // Create or truncate destination file
    int destination_fd = open(destination_file, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (destination_fd == -1) {
        fprintf(stderr, "Error opening file: %s\n", strerror(errno));
        close(source_fd);
        pthread_mutex_unlock(&buf.mutex);
        return;
    }

    // Copy file descriptors and file names to buffer
    buf.buffer[buf.in].source_fd = source_fd;
    buf.buffer[buf.in].destination_fd = destination_fd;
    strcpy(buf.buffer[buf.in].source_file, source_file);
    strcpy(buf.buffer[buf.in].destination_file, destination_file);
    buf.in = (buf.in + 1) % BUFFER_SIZE;
    buf.count++;

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

3-
produce_file_descriptor_pair() function produces a file descriptor pair by opening a source file for reading and a destination file for writing. It adds the file descriptors and file names to a circular buffer (buf) with synchronization using a mutex and condition variables. The function ensures that the buffer is not full before adding the file descriptor pair and signals the availability of data when a file descriptor pair is added to the buffer.

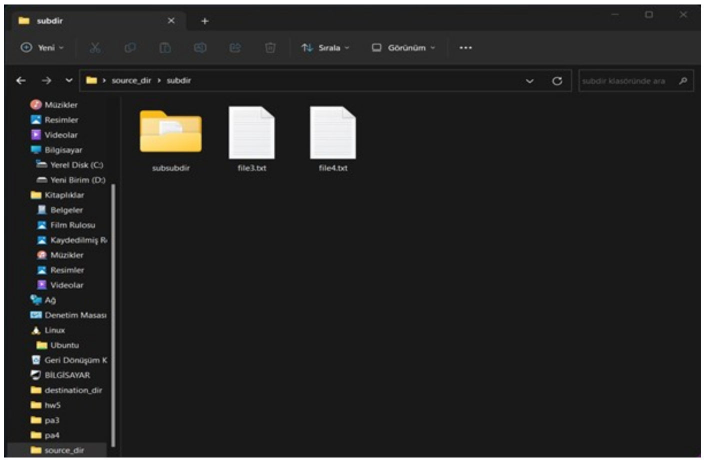


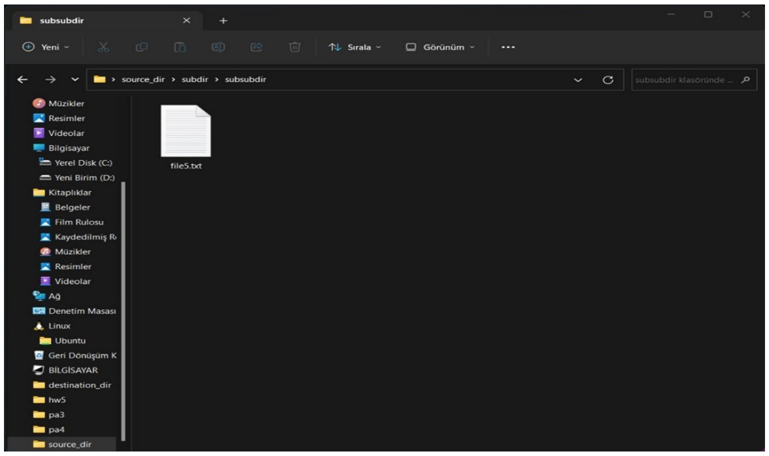
```c
void consume_file_descriptor_pair() {
    pthread_mutex_lock(&buf.mutex);

    // Wait while buffer is empty and producer is not done
    while (buf.count == 0 && !buf.done) {
        pthread_cond_wait(&buf.not_empty, &buf.mutex);
    }

    // Return if buffer is empty and producer is done
    if (buf.count == 0 && buf.done) {
        pthread_mutex_unlock(&buf.mutex);
        return;
    }

    // Get file descriptors and file names from buffer
    int source_fd = buf.buffer[buf.out].source_fd;
    int destination_fd = buf.buffer[buf.out].destination_fd;

    // Copy file content from source to destination
    ssize_t n;
    char buffer[4096];
    while ((n = read(source_fd, buffer, sizeof(buffer))) > 0) {
        write(destination_fd, buffer, n);
    }

    // Close files and print completion status
    close(source_fd);
    close(destination_fd);
    printf("Copied file: %s\n", buf.buffer[buf.out].source_file);

    buf.out = (buf.out + 1) % BUFFER_SIZE;
    buf.count--;

    pthread_cond_signal(&buf.not_full);
    pthread_mutex_unlock(&buf.mutex);
}
```

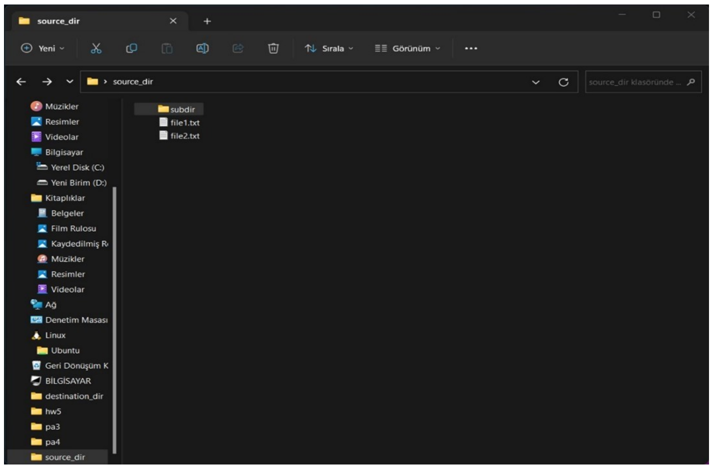
4-

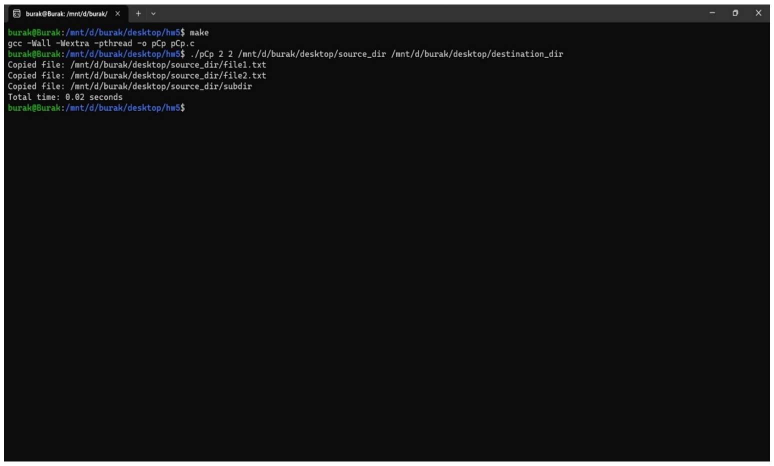
consume_file_descriptor_pair() function consumes a file descriptor pair from the buffer by reading data from the source file descriptor and writing it to the destination file descriptor. It performs synchronization using a mutex and condition variables to ensure that the buffer is not empty before consuming, and it signals the availability of space in the buffer after consuming.

TEST CASE









```
burak@Burak:/mnt/d/burak/desktop/hw5$ make
gcc -Wall -Wextra -pthread -o pCp pCp.c
burak@Burak:/mnt/d/burak/desktop/hw5$ ./pCp 2 2 /mnt/d/burak/desktop/source_dir /mnt/d/burak/desktop/destination_dir
Copied file: /mnt/d/burak/desktop/source_dir/file1.txt
Copied file: /mnt/d/burak/desktop/source_dir/file2.txt
Copied file: /mnt/d/burak/desktop/source_dir/subdir
Total time: 0.02 seconds
burak@Burak:/mnt/d/burak/desktop/hw5$
```