CSE 344 System Programming – Homework 5 Report

General Structure

Initialization:

1. Parse command-line arguments: The program starts by parsing the command-line arguments to get the buffer size, number of worker threads, source directory, and destination directory.
2. Initialize the buffer with the specified size: The shared buffer and synchronization primitives are initialized.
3. Create and start manager and worker threads: The manager thread and worker threads are created and started.

Manager Thread:

1. Traverse the source directory recursively: The manager thread traverses the source directory to find files and directories.For each file, add a FilePair (source and destination paths) to the buffer.
2. Add FilePair to the buffer: For each file found, a FilePair (containing source and destination paths) is added to the buffer.
3. Signal worker threads: The manager thread signals the worker threads whenever a file is added to the buffer.

Worker Threads:

1. Wait for files to be available in the buffer: Worker threads wait until there are files in the buffer.
2. Copy files from source to destination: Worker threads copy files from the source to the destination directory.
3. Update statistics: Worker threads update the statistics on files copied, bytes copied, and errors encountered.

Signal Handling:

1. Handle SIGINT (CTRL+C) for graceful termination: The program handles the SIGINT signal to terminate gracefully.
2. Signal all threads to finish processing and clean up resources: All threads are signaled to finish processing, and resources are cleaned up properly.

Pseudocode

```
MAIN:
  Parse command-line arguments
  Initialize buffer
  Create manager thread
  Create worker threads
  Wait for manager thread to finish
  Wait for worker threads to finish
  Destroy buffer

MANAGER_THREAD:
  Traverse source directory
  For each file:
    Wait if buffer is full (pthread_cond_wait(not_full, mutex))
    Add file pair to buffer
    Signal worker threads (pthread_cond_signal(not_empty))
  Set done flag
  Signal worker threads to finish (pthread_cond_broadcast(not_empty))
  Wait at barrier

WORKER_THREAD:
  Loop:
    Wait if buffer is empty and not done (pthread_cond_wait(not_empty, mutex))
    If buffer is empty and done, exit loop
    Remove file pair from buffer
    Signal manager thread if buffer is not full (pthread_cond_signal(not_full))
    Copy file
    Update statistics
  Wait at barrier

COPY_FILE:
  Read from source file
  Write to destination file
  Update statistics

SIGNAL_HANDLER (SIGINT):
  Set done flag
  Signal all threads to finish (pthread_cond_broadcast(not_empty))
  Wait for threads to finish
  Print statistics
  Destroy buffer and exit
```

Key Points

Thread Usage

- *Manager Thread:* Responsible for traversing directories and adding files to the buffer.
- *Worker Threads:* Multiple threads that handle copying files from the buffer to the destination directory. These threads form a thread pool.

Condition Variables

- *not_empty:* Used to signal worker threads when the buffer has new files added by the manager thread.
- *not_full:* Used to signal the manager thread when space becomes available in the buffer as worker threads remove files from it.

Synchronization: pthread_cond_wait and pthread_cond_signal are used for synchronizing access to the buffer between the manager and worker threads.

Usage in Manager Thread

- *Wait:* The manager thread waits (pthread_cond_wait) on not_full when the buffer is full, indicating it cannot add more files until space is available.
- *Signal:* After adding a file pair to the buffer, the manager thread signals (pthread_cond_signal) the not_empty condition variable to wake up worker threads waiting for new files.

Usage in Worker Threads

- *Wait:* Worker threads wait (pthread_cond_wait) on not_empty when the buffer is empty, indicating there are no files to process.
- *Signal:* After removing a file pair from the buffer, worker threads signal (pthread_cond_signal) the not_full condition variable to wake up the manager thread, indicating space is available.

Buffer Structure

- *Circular Buffer:* Implements a circular queue to store FilePair structures.
- *Critical Section:* Access to the buffer is protected by a mutex to prevent race conditions.

Critical Section

- *Mutex:* pthread_mutex_t is used to ensure exclusive access to shared resources like the buffer and statistics.
- *Protected Code:* All modifications to the buffer and statistics are done within critical sections to prevent data corruption.

Worker Thread Pool

- *Initialization:* Worker threads are created at the start and run concurrently.
- *Task Execution:* Each worker thread waits for a file to be available in the buffer, processes it (copies the file), and updates the statistics.
- *Termination:* Workers continue processing until all files are copied and the done flag is set by the manager thread.

Barrier Synchronization

- *Barrier:* A barrier is used to synchronize the completion of the first phase of processing among all threads.
- *Initialization:* The barrier is initialized to include the manager thread and all worker threads.
- *Manager Thread:* After completing directory traversal and signaling done, the manager thread waits at the barrier.
- *Worker Threads:* Each worker thread waits at the barrier after completing its tasks, ensuring all threads synchronize at this point.

Changes made to Homework 4

- *Barrier Initilization*

Added to `init_buffer` to initialize the barrier.

```
pthread_barrier_init(&buffer.barrier, NULL, config.num_workers + 1);
```

- *Barrier Destruction*

Added to `destroy_buffer` to destroy the barrier.

```
pthread_barrier_destroy(&buffer.barrier);
```

- *Barrier Wait in Manager Thread*

Added to `manager_thread` to wait for all worker threads to finish their first phase.

```
// Wait for all workers to finish their first phase
pthread_barrier_wait(&buffer.barrier);
```

- *Barrier Wait in Worker Threads*

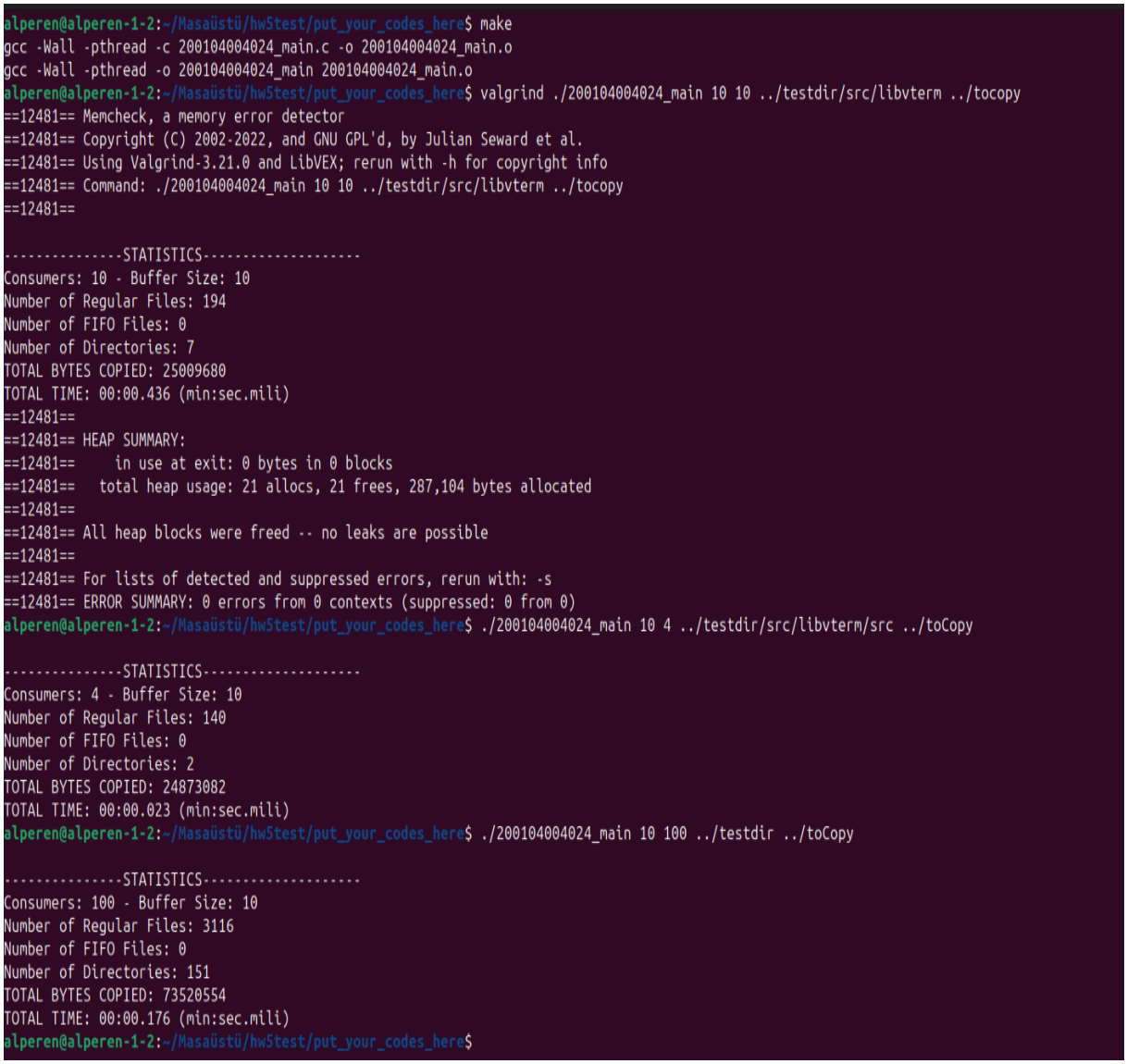
Added to `worker_thread` to wait for all worker threads to finish their first phase.

```
// Wait for all workers to finish their first phase
pthread_barrier_wait(&buffer.barrier);
```

Summary of changes

Ensures that all threads reach a certain point (barrier) before any of them proceed, improving synchronization and coordination among threads.

Screenshots



Alperen Duran 200104004024