Sasken Summer Internship Program - 2025(6th Sem):

Linux-C stream

Project title: File Synchronization Tool

Project Description: Develop a user-space utility that keeps two directories in sync, either locally or across machines, by detecting file changes (create/modify/delete) and propagating updates in near real time. The tool should watch a source directory (e.g., using inotify), maintain a queue of file-system events, and transfer only the changed files using TCP sockets to a target location. It must handle conflict resolution (e.g., newer-timestamp wins), preserve file permissions and metadata, and allow one-way or bi-directional synchronization. This project reinforces Linux file-system monitoring, efficient I/O, socket-based IPC, basic scheduling of periodic scans, and thread safety for concurrent transfers.

Team members :

Ramya	ramyakulal61@gmail.com
Nisha	shettynisha664@gmail.com
Shridevika A S	shridevikaas2004@gmail.com
Sinchana	sinchanah022004@gmail.com
Vidyashree	vidya042004@gmail.com
Pooja Nayak	poojanayak6949@gmail.com
Manisha Suvarna	manishasuvarnam@gmail.com

Table of Contents

1. Introduction
2. Objectives
3. System Architecture and Methodology
4. Implementation Details
5. Testing and Validation
6. Results
7. Conclusion
8. Learning Outcomes
9. References

1. Introduction:

In today's digital environments, data consistency across directories is essential, whether for backup, versioning, or collaborative purposes. This project focuses on building a **user-space file synchronization tool** that ensures two directories remain in sync in near real-time. The tool supports **local one-way synchronization, local bi-directional synchronization**, and **remote one-way synchronization** using TCP sockets. By leveraging Linux file monitoring mechanisms, multithreading, event queues, and inter-process communication, this project provides a robust solution to automatically detect file changes (create, modify, delete) and propagate updates.

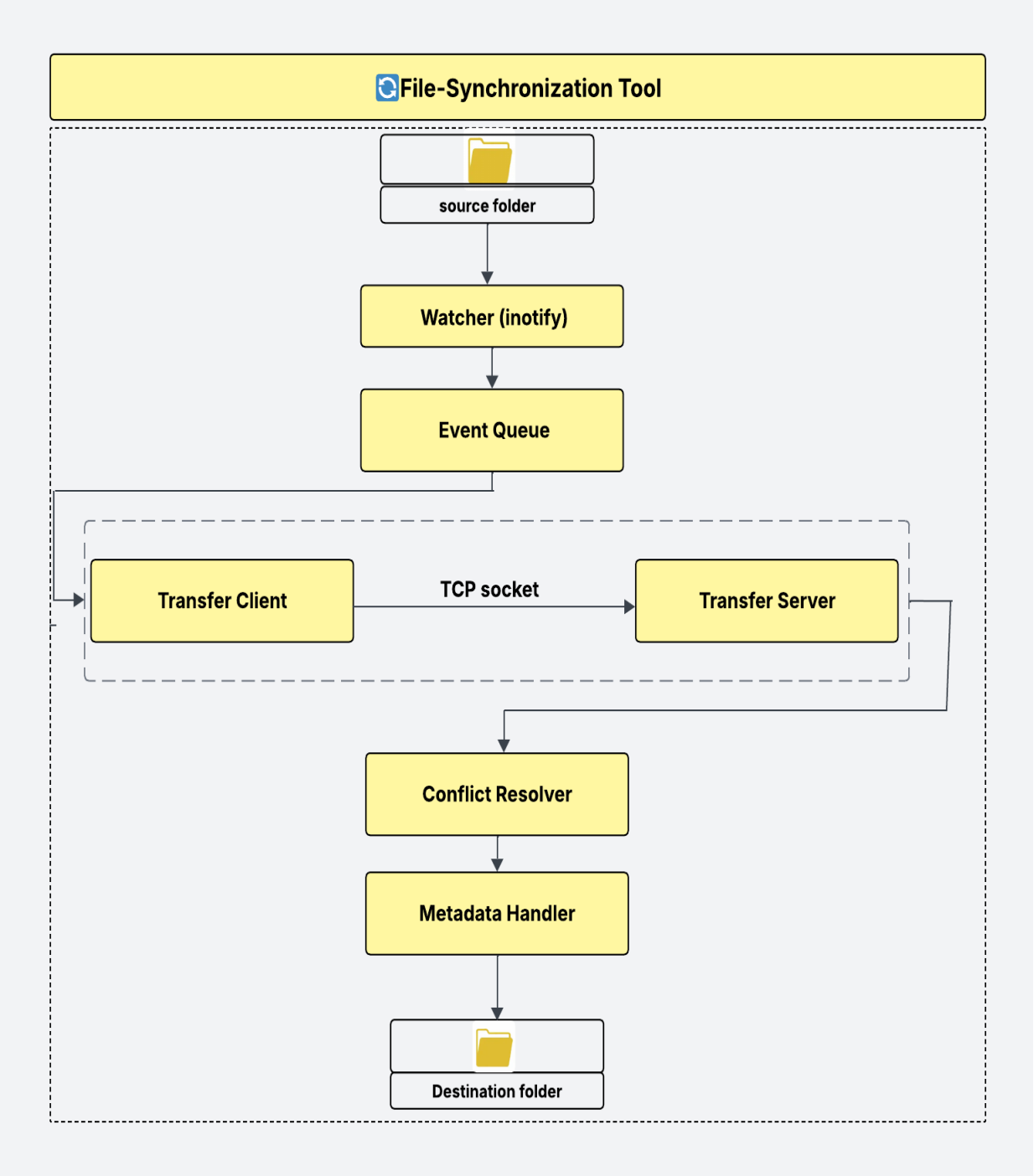
Through this project, we deepened our understanding of **Linux system programming**, including how the file system interacts with processes, memory, and inter-process communication mechanisms. We also learned how to handle concurrency with threads and use sockets to communicate between systems, which is fundamental for real-world distributed synchronization tools.

2. Objectives:

- Build a user-space file synchronization tool for Linux.
- Support local one-way and bi-directional synchronization.
- Implement remote one-way synchronization using TCP sockets.
- Monitor file system changes in real-time using inotify.
- Preserve file permissions and timestamps.
- Use multithreading for concurrent monitoring and synchronization.
- Implement event queues for thread-safe event handling.
- Apply OS concepts like memory management, processes, and scheduling.

3. System Architecture and Methodology

3.1 High-level Workflow



- Parse command-line arguments (source, destination, mode, remote IP, port).
- Initialize event queue and threads (monitor, consumer, scheduler).
- Monitor source directory (and destination for bi-directional) using inotify.

- Push detected file events (create, modify, delete) into the event queue.
- A consumer thread dequeues events and synchronizes files accordingly.
- For remote one-way mode, changes are sent to a server over TCP sockets.
- The scheduler thread periodically rescans directories to ensure consistency.

3.2 Data Structures

1. **Event Queue**: A thread-safe queue to store file system events.

- EventType enum for CREATE, MODIFY, DELETE.
- Linked-list-based queue with pthread_mutex_t and pthread_cond_t for synchronization.

2. **SyncEntry**: Array to track recent modifications for conflict resolution.

3. **SchedulerArgs:** Stores arguments for the background scanner thread.

3.3 Tools and Libraries

1. **Languages**: C

2. **Libraries**: pthread, inotify, fcntl, sys/stat, unistd

3. **Linux APIs**: inotify, pthread, stat, utime, chmod, open, read, write.

4. Implementation Details:

4.1 Source Files

sync_main.c	Main entry point: thread management and flow control.
cli_parser.c/.h	Parses command-line arguments

event_queue.c/.h	Handles thread-safe event queue operations
file_monitor.c/.h	Uses inotify to track file system events
sync_manager.c/.h	Synchronizes files between directories
scheduler.c/.h	Periodic scanner to detect missed changes
Network_tansfer.c/h	Implements TCP server and client for remote one-way sync.
metadata_handler.c/.h	Preserves timestamps and file permissions

4.2 Key Functions

- start_monitoring() – Starts inotify on the directory.
- enqueue_event() – Adds events to the queue.
- dequeue_event() – Processes events from the queue.
- sync_create_or_update() – Copies file with metadata if changed.
- sync_delete() – Deletes a file from destination.
- run_scheduler() – Periodically checks for missed updates.
- start_server() / start_client() – TCP server/client for remote one-way sync.

4.3 Error Handling

- All critical system calls are wrapped with error messages (perror or fprintf(stderr)).
- Invalid CLI options are rejected with clear feedback.
- File open/create operations check for failure.
- Fork failures for remote mode are caught and terminate execution.

4.4 System Calls Used

System calls	Description
inotify_init()	Initializes a new inotify instance to monitor file system events like file creation or deletion.
inotify_add_watch()	Adds a specific file or directory to be watched for changes using inotify.
read()	Reads data from a file descriptor (used to fetch inotify events or file contents).
write()	Writes data to a file descriptor (used for copying file contents to the destination).
open()	Opens a file and returns a file descriptor for reading or writing.
close()	Closes a file descriptor to release the resource.
chmod()	Changes the file permission bits to match the source file's permissions.
fork()	Creates a child process for remote server setup.
stat()	Retrieves file metadata such as size, timestamps, and permission information.
utime()	Updates the access and modification times of a file to preserve original timestamps.
mkdir()	Creates a new directory; used when ensuring destination folders exist.
pthread_create()	Creates a new thread in the current process for multitasking (used for monitoring and syncing).
sleep()	Pauses execution of the current thread for a defined number of seconds (used in the scheduler).

5. Testing and Validation

5.1 Testing Environment

- Platform: Ubuntu via WSL (Windows Subsystem for Linux)
- Directory paths: test/watch_dir and test/target_dir
- Modes : One-way and two-way local synchronization and remote one way synchronization.

5.2 Positive and Negative Test Cases

Test Case Description	Test Type	Expected Output	Result
Create a file in source dir	Positive	File appears in target_dir	Passed
Modify a file in source dir	Positive	Modified file updates in target_dir	Passed
Delete a file in source dir	Positive	File gets removed from target_dir	Passed
Modify file in both source dir and target_dir (two-way)	Positive	Latest timestamp file kept (conflict resolved)	Passed
Remote one-way file creation	Positive	File appears on server target dir	Passed
Delete file directly from target_dir	Negative	File re-copied from source dir	Passed
File created in a different folder	Negative	No synchronization occurs	Passed
Run tool with invalid directory path	Negative	Error message shown, program exits safely	Passed

5.3 Test Procedure

1. Compile using: gcc -std=gnu11 -o build/sync_tool src/*.c -Iinclude -lpthread
2. Prepare directories: mkdir -p test/watch_dir test/target_dir
3. Start one-way sync: ./build/sync_tool --src=test/watch_dir --dest=test/target_dir --mode=onewayCreate/modify/delete files in watch_dir and observe results.
4. Repeat for bi-directional mode using --mode=twoway.
5. For one way remote synchronization:

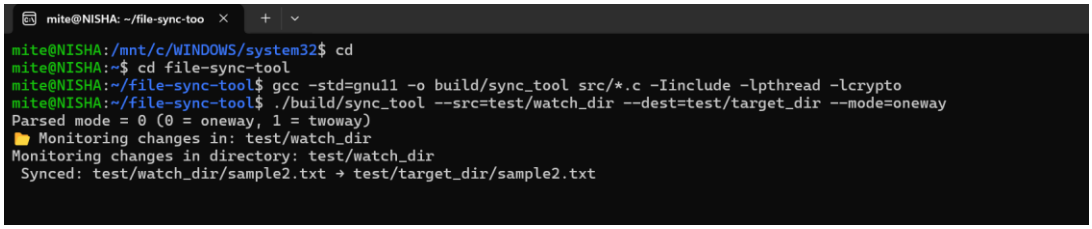
On server: ./build/sync_tool --src=test/target_dir --dest=test/target_dir --remote --mode=oneway --port=8080

On client: ./build/sync_tool --src=test/watch_dir --dest=test/target_dir --remote --mode=oneway --port=8080 --ip=<server-ip>

6. Create file in client watch dir → file appears on server.

6. Results

Local synchronization: One-way



Local synchronization: Bi-directional

Remote synchronization(Across machine):One-way

click_here_remote_synchronization

Snippet 1: Thread Creation for Monitor, Scheduler, and Consumer

```
pthread_t monitor_thread, scheduler_thread, consumer_thread;
pthread_create(&monitor_thread, NULL, run_monitoring, NULL);
pthread_create(&scheduler_thread, NULL, run_scheduler, NULL);
pthread_create(&consumer_thread, NULL, run_consumer, NULL);
```

Initializes and runs three core threads: monitoring file events, scanning directories periodically, and processing the sync queue.

Snippet 2: sync_create_or_update() – Synchronizing Files

```
int src_fd = open(src_path, O_RDONLY);
int dst_fd = open(dst_path, O_WRONLY | O_CREAT | O_TRUNC,
src_stat.st_mode);
while ((bytes = read(src_fd, buf, sizeof(buf))) > 0)
      write(dst_fd, buf, bytes);
```

Performs file copy from source to destination, used during sync. Ensures correct data transfer.

Snippet 3: Periodic Rescan by the Scheduler

```
      void* run_scheduler(void* arg) {
         while (1) {
            scan_directory(config.src_path, config.dest_path, &queue);
            if (config.mode == MODE_TWOWAY)
               scan_directory(config.dest_path, config.src_path, &queue);
            sleep(5);
         }
      }
```

Ensures consistency by periodically scanning both source and destination directories.

Snippet 4: Inotify Setup to Detect File Events

```
   int wd = inotify_add_watch(inotify_fd, path, IN_CREATE | IN_MODIFY |
IN_DELETE);
length = read(inotify_fd, buffer, EVENT_BUF_LEN);
for (i = 0; i < length;) {
   struct inotify_event *event = (struct inotify_event *) &buffer[i];
}
```

Detects file creation, modification, and deletion in real time using inotify, and pushes events into a queue.

Snippet 5: Handling File Deletion

```
void sync_delete(const char *dst_path) {
   printf("Deleting %s\n", dst_path);
   if (remove(dst_path) == 0)
      printf(" Deleted: %s\n", dst_path);
}
```

Removes files from the destination directory if they are deleted in the source, maintaining sync integrity.

Snippet 6: Metadata Preservation

```
struct stat src_stat;
stat(src_path, &src_stat);
chmod(dst_path, src_stat.st_mode);
struct utimbuf times = {src_stat.st_atime, src_stat.st_mtime};
utime(dst_path, &times);
```

Preserves file permissions and modification timestamps after syncing, ensuring metadata consistency.

7. Conclusion

This file synchronization tool successfully demonstrates the ability to keep two local directories in sync, both in one-way and bi-directional modes and one way remote synchronization. Using Linux-specific features like inotify, threading, and system calls, the tool offers real-time synchronization with efficient conflict resolution. The modular architecture and extensible design also lay the groundwork for potential remote synchronization features in the future. Despite time constraints, the project showcases key concepts of operating systems through a practical and working solution.

8. Learning Outcomes

- **File System Monitoring:** Gained hands-on experience with inotify.

- **Multithreading:** Learned producer-consumer synchronization with pthread.
- **Scheduling**: Added periodic rescanning to ensure sync integrity.
- **Metadata Handling:** Preserved file permissions and timestamps.
- **Memory Management:** Managed dynamically allocated event queues.
- **Processes:** Used fork for server creation in remote sync.
- **Socket Programming:** Implemented TCP-based remote one-way synchronization.

OS Concepts Implemented:

1. Memory Management
2. Virtual Memory
3. Processes
4. Scheduling
5. Multithreading
6. Inter-process Communication (via sockets)
7. Socket Programming
8. File System

9. References

- inotify(7) - Linux manual
- pthread Library Documentation
- Linux stat, chmod, utime, read, write system calls
- GeeksforGeeks - Multithreading in C
- GNU C Library Documentation
- OS lab syllabus