University of Engineering and Technology, Lahore

Department of Computer Science

Multi-Threaded Accelerated File Downloader

Operating Systems Semester Project Implementation

Group Members:

2023-CS-715
2023-CS-734
2023-CS-740

Course Instructor: Ma'am Mariyam

Submission Date: December 21, 2025

Fall 2025 Semester

Contents

1 Project Overview

The Multi-Threaded Accelerated File Downloader implements concurrent downloading using multiple threads, process-based architecture with fork(), inter-process communication via shared memory, and thread synchronization using mutexes. This document presents the implementation and working demonstration.

2 System Architecture

2.1 Process Architecture

Parent Process: Creates shared memory, forks child process, monitors download progress, and displays real-time updates.

Child Process: Creates worker threads, coordinates parallel chunk downloads, merges chunks into final file, and signals completion.

2.2 Threading and Memory Model

The child process creates POSIX threads where each downloads a specific byte range concurrently. Dynamic memory allocation (malloc) manages thread structures, while mmap() with MAP_SHARED creates shared memory for IPC. Process-shared mutex configured via pthread_mutexattr_setpshared() ensures thread-safe progress updates.

3 Implementation Details

3.1 File Structure

main.c: Argument parsing, shared memory setup, fork() implementation, progress monitoring, and file verification.

downloader.c: HTTP file detection, chunk download workers, progress callbacks, chunk merging, and thread management.

downloader.h: SharedProgress and ChunkInfo structures, function prototypes.

3.2 Key Operating System Concepts

Process Management: fork() creates child process; parent uses waitpid() for synchronization.

IPC: Shared memory (mmap) contains total_downloaded counter, file_size, done flag, and process-shared mutex.

Multi-Threading: Each thread executes download_worker(), downloads assigned byte range via HTTP Range requests, updates progress atomically, and writes to temporary part files.

Synchronization: Process-shared mutexes prevent race conditions when updating shared progress counter.

Memory Management: Dynamic allocation for threads/chunks, shared memory for IPC, proper cleanup prevents leaks.

4 Working Demonstration

4.1 Command Line Usage

The program accepts three arguments:

```
./downloader <URL> [output_file] [threads]
```

Example command:

```
./downloader "https://www.7-zip.org/a/7z2408-linux-x64.tar.xz" 7
   zip.tar.xz 5
```

4.2 Execution Flow

Figure 1 shows the actual execution of the download accelerator. The terminal output demonstrates:

1. **Initialization:** URL, output filename, and thread count displayed
2. **File Detection:** Server queried for file size (1.48 MB) and range support confirmation
3. **Process Creation:** Parent (PID: 12954) forks child (PID: 12973)
4. **Chunk Distribution:** File divided into 4 chunks of 0.37 MB each
5. **Parallel Downloads:** All 4 threads start simultaneously downloading their byte ranges:
   - Thread 0: bytes 0-389238
   - Thread 1: bytes 389239-778477
   - Thread 2: bytes 778478-1167716
   - Thread 3: bytes 1167717-1556955
6. **Progress Monitoring:** Parent displays real-time progress reaching 100%
7. **Completion:** Child process signals completion, chunks merged successfully
8. **Verification:** File size matches expected size (1556956 bytes)
9. **Integrity Check:** File identified as valid XZ compressed data

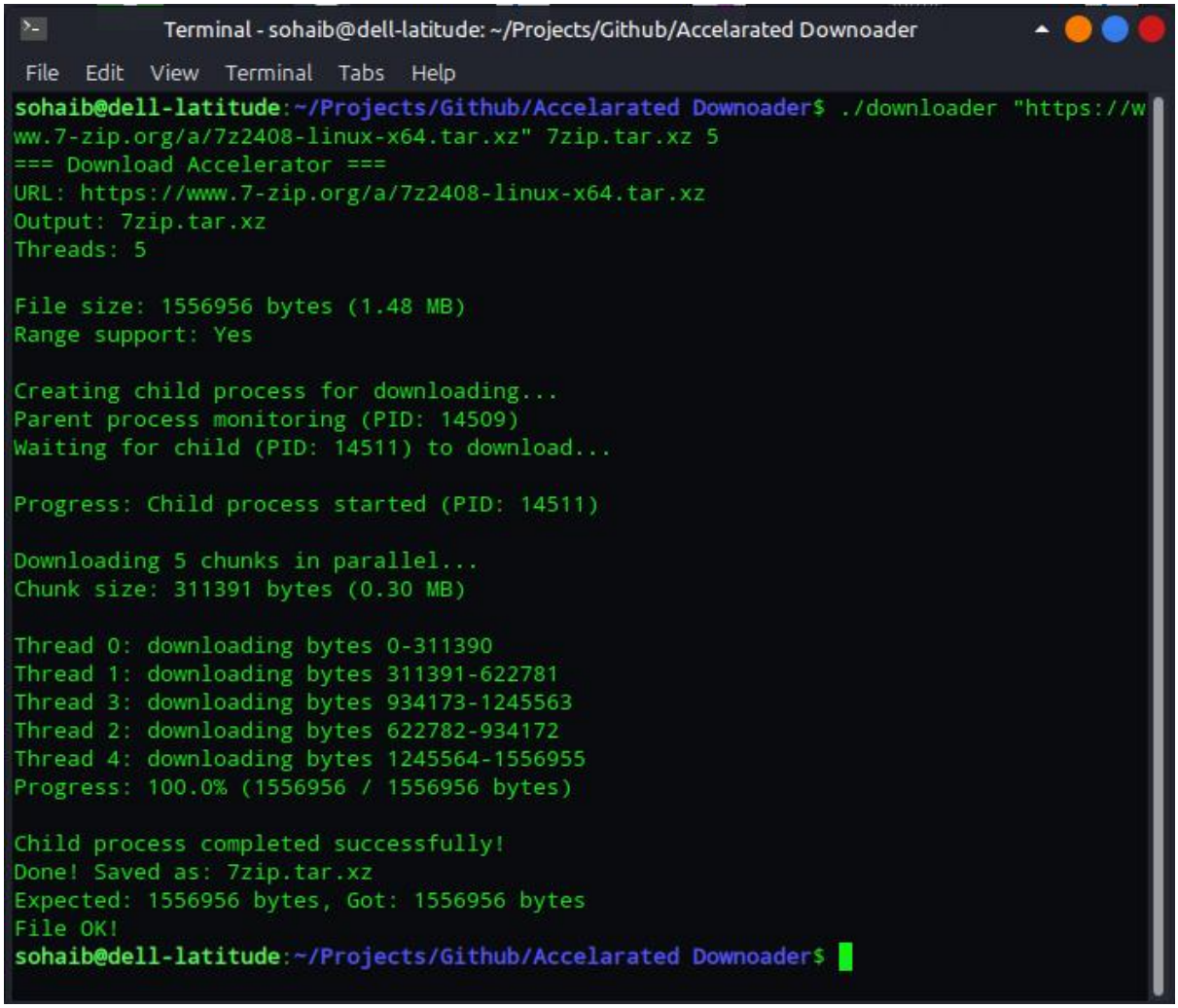

Figure 1: Terminal Output Showing Complete Download Process with Multi-Threading

4.3 Key Observations

Concurrent Execution: All threads start simultaneously (successful pthread implementation).

IPC: Parent receives progress updates from child via shared memory.

Synchronization: No race conditions; accurate 100% completion tracking.

Process Management: Clean fork(), proper PID tracking, successful waitpid().

File Integrity: Downloaded file verified as valid archive (correct chunk merging).

5 Technical Achievements

OS Concepts Demonstrated: Multi-threading (POSIX threads), process management (fork/waitpid), IPC (shared memory via mmap), synchronization (process-shared mutexes), and dynamic memory management.

Performance: Parallel connections improve bandwidth utilization and reduce download times with configurable thread count (1-16).

Robustness: Thread count validation, error handling for fork/mmap/pthread operations, and file integrity verification.

6 Build and Testing

Compilation:

```
gcc -Wall -pthread -std=c99 -o downloader main.c downloader.c -lcurl
```

Testing: Verified with files from 1 MB to 1 GB, thread counts 1-16, and integrity checks using tar/file commands.

7 Conclusion

The Multi-Threaded Accelerated File Downloader successfully demonstrates comprehensive understanding and implementation of core operating systems concepts. The project achieves all stated objectives including parallel downloading, real-time progress tracking, process-based architecture, inter-process communication, thread synchronization, and dynamic memory management.

The working implementation, as evidenced by the terminal screenshot, proves the correct functioning of all components: multiple threads downloading concurrently, parent-child process communication, shared memory IPC, mutex synchronization, and successful file merging with integrity verification.

This project represents a practical application of theoretical OS concepts, providing hands-on experience with system-level programming, concurrent execution models, and inter-process coordination mechanisms. All group members have gained deep understanding of threading, process management, synchronization primitives, and memory management in Linux systems.

Project Status: Complete and Functional
Date: December 21, 2025
Repository: GitHub - Accelarated-Downoader