CS 3502 Project 1 - Multi-Threaded Programming and IPC

Author Name
Section W03
pnsajja@students.kennesaw.edu

February 28, 2025

Contents

1 Introduction

This project demonstrates fundamental operating system concepts by implementing multi-threaded programming and inter-process communication (IPC) in C on a Linux system. The project is divided into two main parts:

1. **Multi-Threading Implementation (Project A):** A banking system simulation is implemented in four phases:

Phase 1: Basic Thread Operations: Threads are created to perform independent operations, demonstrating basic concurrency and thread management.

Phase 2: Resource Protection: Mutex locks are used to synchronize access to shared bank accounts, ensuring that concurrent operations do not result in race conditions.

Phase 3: Deadlock Creation: A natural deadlock scenario is demonstrated by allowing threads to acquire locks in inconsistent orders during multi-account transfers.

Phase 4: Deadlock Resolution: Deadlock prevention and recovery are achieved using a timeout mechanism and consistent lock ordering.

2. **Inter-Process Communication (Project B):** A custom IPC program processes the output of a command (e.g., `ls -l`) by using a pipe to transfer data from a child process (which executes the command) to a parent process (which reads and processes the output).

Comprehensive testing and logging have been incorporated into both parts to validate functionality, performance, and error handling.

2 Objectives and Scope

The primary objectives of this project are:

- To implement multi-threaded programming with proper thread creation, synchronization, and management.
- To demonstrate techniques for preventing and resolving deadlocks in a multi-threaded environment.
- To implement inter-process communication using pipes for data transfer between processes.
- To validate the implementations with unit tests and detailed logging.
- To document the design, implementation, and testing processes comprehensively.

3 Implementation Details

3.1 Multi-Threading Implementation (Project A)

The multi-threading component simulates a banking system with multiple accounts and threads performing transactions. It is divided into four phases:

3.1.1 Phase 1: Basic Thread Operations

In this phase, two threads perform a deposit and a withdrawal on the same account without any mutex locks, demonstrating how a race condition can occur.

Example Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>

// Bank account structure with an ID, balance, and associated mutex
typedef struct {
    int id;
    long balance;
    pthread_mutex_t mutex;
} Account;

#define NUM_ACCOUNTS 5
Account accounts[NUM_ACCOUNTS];

// Parameter structure for deposit/withdraw operations
typedef struct {
    int thread_id;
    int acc_id;
    long amount;       // positive for deposit, negative for withdraw
    int use_mutex;     // flag to indicate if mutex locking should be used
} OpParams;

// The function used in Phase 1 (no mutex locking)
void* perform_operation(void* arg) {
    OpParams* p = (OpParams*) arg;
    int tid = p->thread_id;
    int acc = p->acc_id;
    long amount = p->amount;
    const char* opType = (amount >= 0) ? "Deposit" : "Withdraw";

    printf("[Thread %d] %s \$%ld on Account %d (starting)\n", tid, opType,
           (amount >= 0 ? amount : -amount), acc);

    // No locking for Phase 1
    long old_balance = accounts[acc].balance;
    usleep(100000);  // simulate work to expose race conditions
    long new_balance = old_balance + amount;
    accounts[acc].balance = new_balance;
    printf("[Thread %d] updated Account %d balance: %ld -> %ld\n",
           tid, acc, old_balance, new_balance);

    printf("[Thread %d] %s on Account %d (completed)\n", tid, opType, acc);
    pthread_exit(NULL);
}
```

3.1.2 Phase 2: Resource Protection

In Phase 2, the same function `perform_operation` is used, but with the `use_mutex` flag enabled, so each thread locks the account before modifying it, preventing race conditions.

Example Code:

```c
void* perform_operation(void* arg) {
    OpParams* p = (OpParams*) arg;
    int tid = p->thread_id;
    int acc = p->acc_id;
    long amount = p->amount;
    const char* opType = (amount >= 0) ? "Deposit" : "Withdraw";

    printf("[Thread %d] %s \$%ld on Account %d (starting)\n", tid, opType,
           (amount >= 0 ? amount : -amount), acc);

    // Lock the account mutex if synchronization is enabled
    if (p->use_mutex) {
        pthread_mutex_lock(&accounts[acc].mutex);
        printf("[Thread %d] acquired lock on Account %d\n", tid, acc);
    }

    // Critical section
    long old_balance = accounts[acc].balance;
    usleep(100000);
    long new_balance = old_balance + amount;
    accounts[acc].balance = new_balance;
    printf("[Thread %d] updated Account %d balance: %ld -> %ld\n",
           tid, acc, old_balance, new_balance);

    // Unlock mutex if it was locked
    if (p->use_mutex) {
        pthread_mutex_unlock(&accounts[acc].mutex);
        printf("[Thread %d] released lock on Account %d\n", tid, acc);
    }

    printf("[Thread %d] %s on Account %d (completed)\n", tid, opType, acc);
    pthread_exit(NULL);
}
```

3.1.3 Phase 3: Deadlock Creation

In Phase 3, two threads transfer money in opposite directions without any deadlock avoidance, causing them to lock different accounts and wait on each other indefinitely.

Example Code:

```c
typedef struct {
    int thread_id;
    int from_acc;
    int to_acc;
    long amount;
    int use_deadlock_avoidance;
} TransferParams;

// Phase 3: naive locking, prone to deadlock
void* perform_transfer(void* arg) {
    TransferParams* p = (TransferParams*) arg;
    int tid = p->thread_id;
    int from = p->from_acc;
    int to   = p->to_acc;
    long amount = p->amount;

    printf("[Thread %d] Transfer \$%ld from Account %d to Account %d (starting)\n"
    ,
```

```
            tid, amount, from, to);

    if (!p->use_deadlock_avoidance) {
        // Lock 'from' first, then 'to'
        pthread_mutex_lock(&accounts[from].mutex);
        printf("[Thread %d] locked Account %d, now trying to lock Account %d\n",
               tid, from, to);
        usleep(100000);
        pthread_mutex_lock(&accounts[to].mutex);
        printf("[Thread %d] locked Account %d\n", tid, to);

        // Perform transfer
        accounts[from].balance -= amount;
        accounts[to].balance   += amount;
        printf("[Thread %d] transferred \$%ld (Account %d new balance: %ld, "
               "Account %d new balance: %ld)\n",
               tid, amount, from, accounts[from].balance, to, accounts[to].balance
    );

        // Release locks
        pthread_mutex_unlock(&accounts[to].mutex);
        pthread_mutex_unlock(&accounts[from].mutex);
        printf("[Thread %d] Transfer completed and locks released\n", tid);
    } else {
        // Phase 4 code will go here
    }

    printf("[Thread %d] Transfer from Account %d to %d (finished)\n", tid, from,
    to);
    pthread_exit(NULL);
}
```

3.1.4 Phase 4: Deadlock Resolution

A consistent lock ordering (lower-numbered account first) plus a timeout mechanism avoids deadlocks. If the second lock cannot be acquired quickly, the thread releases the first lock and retries later.

Example Code:

```
else {
    // Phase 4: deadlock avoidance with ordered locking and retry
    int first = p->from_acc;
    int second = p->to_acc;
    if (first > second) {
        first = p->to_acc;
        second = p->from_acc;
    }

    int got_first = 0, got_second = 0;
    const int max_retries = 5;
    int attempt = 0;

    while (attempt < max_retries) {
        attempt++;
        if (!got_first) {
            pthread_mutex_lock(&accounts[first].mutex);
            got_first = 1;
```

```c
            printf("[Thread %d] locked Account %d (first lock)\n", p->thread_id, first);
        }
        if (pthread_mutex_trylock(&accounts[second].mutex) == 0) {
            got_second = 1;
            printf("[Thread %d] locked Account %d (second lock)\n", p->thread_id, second);
        } else {
            printf("[Thread %d] could not lock Account %d (held by another thread). "
                   "Releasing Account %d and retrying...\n",
                   p->thread_id, second, first);
            pthread_mutex_unlock(&accounts[first].mutex);
            got_first = 0;
            usleep(100000 + (rand() % 100000));
            continue;
        }
        if (got_first && got_second) break;
    }

    if (got_first && got_second) {
        accounts[p->from_acc].balance -= p->amount;
        accounts[p->to_acc].balance   += p->amount;
        printf("[Thread %d] transferred \$%ld from Account %d to %d (new balances: %ld, %ld)\n",
               p->thread_id, p->amount, p->from_acc, p->to_acc,
               accounts[p->from_acc].balance, accounts[p->to_acc].balance);
        pthread_mutex_unlock(&accounts[second].mutex);
        pthread_mutex_unlock(&accounts[first].mutex);
        printf("[Thread %d] Transfer completed and locks released\n", p->thread_id);
    } else {
        if (got_first) pthread_mutex_unlock(&accounts[first].mutex);
        printf("[Thread %d] Transfer aborted to avoid deadlock\n", p->thread_id);
    }
}
```

3.1.5 Unit Testing for Multi-Threading

A main function initializes the accounts, runs each phase, and prints the results, confirming whether race conditions, deadlocks, or successful transfers occur as expected.

Main Test Runner for Multi-Threading:

```c
int main() {
    srand(time(NULL));
    printf("===== Project A: Multi-Threading Implementation =====\n");
    printf("Initializing %d accounts with \$100 each.\n", NUM_ACCOUNTS);
    for (int i = 0; i < NUM_ACCOUNTS; ++i) {
        accounts[i].id = i;
        accounts[i].balance = 100;
        pthread_mutex_init(&accounts[i].mutex, NULL);
    }

    /*** Phase 1: Basic Thread Operations ***/
    printf("\n---- Phase 1: Basic Thread Operations (No Mutex) ----\n");
    accounts[0].balance = 100;
    printf("Account 0 initial balance: %ld\n", accounts[0].balance);
    OpParams p1 = {1, 0, 50, 0};   // deposit \$50
```

```c
    OpParams p2 = {2, 0, -50, 0};  // withdraw \$50
    pthread_t t1, t2;
    pthread_create(&t1, NULL, perform_operation, &p1);
    pthread_create(&t2, NULL, perform_operation, &p2);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Account 0 final balance: %ld (expected 100)\n", accounts[0].balance);
    if (accounts[0].balance != 100) {
        printf("** Race condition observed! Expected 100, got %ld **\n", accounts[0].balance);
    } else {
        printf("No race condition observed.\n");
    }

    /*** Phase 2: Resource Protection with Mutexes ***/
    printf("\n---- Phase 2: Resource Protection (Using Mutexes) ----\n");
    for (int i = 0; i < NUM_ACCOUNTS; ++i) {
        accounts[i].balance = 100;
    }
    printf("All accounts reset to \$100.\n");
    pthread_t threads[NUM_ACCOUNTS * 2];
    OpParams params[NUM_ACCOUNTS * 2];
    int tid_counter = 3;
    for (int i = 0; i < NUM_ACCOUNTS; ++i) {
        params[2*i]   = (OpParams){ tid_counter++, i, 50, 1 };   // deposit \$50
        params[2*i+1] = (OpParams){ tid_counter++, i, -50, 1 };  // withdraw \$50
        pthread_create(&threads[2*i],   NULL, perform_operation, &params[2*i]);
        pthread_create(&threads[2*i+1], NULL, perform_operation, &params[2*i+1]);
    }
    for (int i = 0; i < NUM_ACCOUNTS * 2; ++i) {
        pthread_join(threads[i], NULL);
    }
    long total_balance = 0;
    for (int i = 0; i < NUM_ACCOUNTS; ++i) {
        total_balance += accounts[i].balance;
        printf("Account %d final balance: %ld (expected 100)\n", i, accounts[i].balance);
    }
    printf("Total balance across all accounts: %ld (expected %d)\n", total_balance, NUM_ACCOUNTS * 100);
    if (total_balance != NUM_ACCOUNTS * 100) {
        printf("** Discrepancy in total balance detected! **\n");
    } else {
        printf("All account balances correct. Mutex synchronization successful.\n");
    }

    /*** Phase 3: Deadlock Creation ***/
    printf("\n---- Phase 3: Deadlock Creation ----\n");
    accounts[0].balance = 100;
    accounts[1].balance = 100;
    printf("Account 0 balance = %ld, Account 1 balance = %ld\n", accounts[0].balance, accounts[1].balance);
    TransferParams tp1 = {1, 0, 1, 30, 0};  // Thread 1: transfer \$30 from Account 0->1
    TransferParams tp2 = {2, 1, 0, 20, 0};  // Thread 2: transfer \$20 from Account 1->0
    pthread_t td1, td2;
    pthread_create(&td1, NULL, perform_transfer, &tp1);
```

```
    pthread_create(&td2, NULL, perform_transfer, &tp2);
    sleep(1);  // allow time for deadlock
    printf("Deadlock likely occurred (threads are waiting on each other).\n");
    printf("Proceeding to Phase 4 to resolve deadlock...\n");
    pthread_cancel(td1);
    pthread_cancel(td2);
    pthread_mutex_destroy(&accounts[0].mutex);
    pthread_mutex_destroy(&accounts[1].mutex);
    pthread_mutex_init(&accounts[0].mutex, NULL);
    pthread_mutex_init(&accounts[1].mutex, NULL);

    /*** Phase 4: Deadlock Resolution ***/
    printf("\n---- Phase 4: Deadlock Resolution ----\n");
    accounts[0].balance = 100;
    accounts[1].balance = 100;
    printf("Account 0 balance = %ld, Account 1 balance = %ld\n", accounts[0].
   balance, accounts[1].balance);
    TransferParams tp3 = {3, 0, 1, 30, 1};  // Thread 3: transfer \$30 w/
   avoidance
    TransferParams tp4 = {4, 1, 0, 20, 1};  // Thread 4: transfer \$20 w/
   avoidance
    pthread_create(&td1, NULL, perform_transfer, &tp3);
    pthread_create(&td2, NULL, perform_transfer, &tp4);
    pthread_join(td1, NULL);
    pthread_join(td2, NULL);
    printf("After transfers: Account 0 = %ld, Account 1 = %ld, Total = %ld (
   expected 200)\n",
           accounts[0].balance, accounts[1].balance, accounts[0].balance +
   accounts[1].balance);
    if (accounts[0].balance + accounts[1].balance != 200) {
        printf("** Total balance inconsistency detected! **\n");
    }
    printf("Multi-threading demonstration completed.\n");

    return 0;
}
```

Listing 1: Main Test Runner for Multi-Threading

3.2 Inter-Process Communication (Project B)

The IPC program processes the output of the `ls -l` command by creating a pipe and forking a child process that executes `ls -l`. The parent process reads from the pipe, line by line, and computes file statistics.

Example Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>
#include <sys/time.h>
#include <signal.h>

int main(int argc, char* argv[]) {
    // Determine target directory for ls. If an argument is provided, use it;
   otherwise use current directory.
    const char *targetDir = (argc > 1 ? argv[1] : ".");
```

```
    printf("===== Project B: IPC (Pipe) Implementation =====\n");
    printf("Executing 'ls -l %s' and processing output...\n", targetDir);

    // Create pipe
    int pipefd[2];
    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    // Fork child process
    pid_t pid = fork();
    if (pid < 0) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (pid == 0) {
        /*** Child Process ***/
        close(pipefd[0]);              // close unused read end
        dup2(pipefd[1], STDOUT_FILENO); // redirect stdout to pipe write end
        close(pipefd[1]);              // close original write descriptor

        // Execute "ls -l <targetDir>"
        execlp("ls", "ls", "-l", targetDir, (char*)NULL);
        // If exec fails, print error and exit
        perror("execlp ls");
        _exit(1);
    }
    else {
        /*** Parent Process ***/
        close(pipefd[1]);  // close unused write end of pipe

        // Open the pipe read end as a FILE* stream for easier reading
        FILE *stream = fdopen(pipefd[0], "r");
        if (!stream) {
            perror("fdopen");
            exit(EXIT_FAILURE);
        }

        struct timeval start, end;
        gettimeofday(&start, NULL);

        // Variables for parsing and statistics
        char *line = NULL;
        size_t len = 0;
        ssize_t nread;
        long long total_bytes = 0;      // total bytes read from pipe
        int total_entries = 0;          // total items (files + directories)
        int file_count = 0;
        int dir_count = 0;
        long long total_file_size = 0;  // sum of sizes of files

        // Read and process each line from the pipe
        while ((nread = getline(&line, &len, stream)) != -1) {
            total_bytes += nread;
            // Remove trailing newline for easier parsing
            if (nread > 0 && line[nread - 1] == '\n') {
                line[nread - 1] = '\0';
```

```c
            }
            if (strlen(line) == 0) {
                continue; // skip empty lines
            }
            if (strncmp(line, "total ", 6) == 0) {
                continue; // skip the "total N" line from ls -l output
            }

            // Parse the line: format "perm links owner group size date name"
            char perms[16];
            long long size = 0;
            perms[0] = '\0';
            int items = sscanf(line, "%15s %*s %*s %*s %lld", perms, &size);
            if (items < 2) {
                fprintf(stderr, "Warning: Unrecognized line format, skipping: %s\n", line);
                continue;
            }

            // Count entry and categorize by type
            total_entries++;
            if (perms[0] == 'd') {
                dir_count++;
            } else {
                file_count++;
                total_file_size += size;
            }
        }

        // Reading done; clean up
        free(line);
        fclose(stream);

        gettimeofday(&end, NULL);
        // Calculate elapsed time
        double elapsed_sec = (end.tv_sec - start.tv_sec)
                           + (end.tv_usec - start.tv_usec) / 1000000.0;

        // Wait for child process to finish and get exit status
        int status;
        waitpid(pid, &status, 0);
        if (WIFSIGNALED(status)) {
            fprintf(stderr, "Error: Child process terminated by signal %d\n", WTERMSIG(status));
            if (WTERMSIG(status) == SIGPIPE) {
                fprintf(stderr, "Broken pipe: Child received SIGPIPE (no reader)\n");
            }
        } else if (WIFEXITED(status) && WEXITSTATUS(status) != 0) {
            fprintf(stderr, "Error: Child process exited with status %d (ls command failed?)\n",
                    WEXITSTATUS(status));
        }

        // Output the processed results
        printf("Total entries: %d (%d files, %d directories)\n", total_entries, file_count, dir_count);
        printf("Total size of files: %lld bytes\n", total_file_size);
        if (elapsed_sec < 1e-6) elapsed_sec = 1e-6; // avoid division by zero
```

```
        double throughput = (total_bytes / 1048576.0) / elapsed_sec;
        printf("Data read: %lld bytes in %.4f seconds (%.2f MB/s)\n",
               total_bytes, elapsed_sec, throughput);
    }

    return 0;
}
```

4 Environment Setup and Tool Usage

The project was developed on Ubuntu running in VirtualBox. GCC was used as the compiler, and Visual Studio Code was used for code editing. The development environment required installing necessary packages (such as build-essential and pthreads), and any issues were resolved by referring to online documentation and community resources.

5 Challenges and Solutions

Several challenges were encountered:

- **Race Conditions:** Unsynchronized access in Phase 1 led to inconsistent results. This was resolved by using mutexes in Phase 2.
- **Deadlock Creation:** Natural deadlocks were difficult to induce reliably. Randomized locking orders and deliberate delays (using `usleep`) were introduced to create a deadlock in Phase 3.
- **Deadlock Resolution:** Implementing a timeout mechanism and enforcing a consistent locking order in Phase 4 required careful design. A back-off strategy was added to ensure that threads could recover from potential lock acquisition failures.
- **IPC Parsing:** Processing the output of `ls -l` required robust parsing to handle various formats. Standard I/O functions and careful error handling ensured that the output was processed correctly.

6 Results and Outcomes

The following outcomes were achieved:

- **Multi-Threading:**
  - Phase 1 demonstrated thread creation and concurrent execution, though race conditions were evident.
  - Phase 2 successfully synchronized access using mutexes, yielding consistent account balances.
  - Phase 3 produced a natural deadlock, as indicated by halted progress and logged messages.
  - Phase 4 resolved deadlock issues through timeout and ordered locking, ensuring that all transfers completed (or were safely skipped) and total balances remained correct.
- **IPC:**

- The IPC program processed the output of the `ls -l` command accurately.
- Data integrity was maintained, and performance metrics (elapsed time and throughput) were within acceptable ranges.
- Error handling mechanisms functioned as expected.

7 Reflection and Learning

The project provided valuable practical experience in implementing multi-threading and IPC. Key learning points include:

- Effective use of POSIX Threads for concurrent programming.
- Importance of synchronization to prevent race conditions.
- Understanding how deadlocks occur and methods to resolve them.
- Implementing inter-process communication using pipes, including robust error handling.
- The significance of thorough testing and detailed logging in diagnosing concurrency issues.

8 References

- Linux Man Pages. Available at: `https://man7.org/linux/man-pages/`
- GeeksforGeeks. *Multithreading in C Using Pthreads*. Available at: `https://www.geeksforgeeks.org/multithreading-in-c-2/`
- TutorialsPoint. *Deadlock in Operating Systems*. Available at: `https://www.tutorialspoint.com/operating_system/os_deadlock.htm`
- The Geek Stuff. *Inter-Process Communication (IPC) Using Pipes in Linux*. Available at: `https://www.thegeekstuff.com/2012/02/ipc-pipes/`
- Stack Overflow. *How to avoid deadlock in multi-threaded C programs?*. Available at: `https://stackoverflow.com/questions/4231497/how-do-i-avoid-deadlock-in-multi-threaded-c-program`