Operating System Architecture Project Report

Turan Abbasli and Sanan Aslanov

Abstract

This report documents the development of a bilingual word translation server application as part of the Operating System Architecture project. Implemented in three versions, the project progressively incorporates advanced features such as shared memory, message queues, and threading. It demonstrates inter-process communication (IPC) techniques in C and showcases a practical system for dynamic word translation.

Contents

1 Introduction

This project aimed to develop a server application for bilingual word translation between English and French. The system dynamically updates its dictionary by reading files containing English-French word pairs. The project highlights the use of inter-process communication (IPC) techniques, including signal handling, shared memory, and message queues, to achieve efficient real-time translation.

2 Project Description

2.1 Objective

The primary objective was to create a translation server capable of providing real-time translations while dynamically updating its dictionary. Each version of the project progressively introduced more sophisticated features.

2.2 File Structure

The dictionary files are formatted as follows:

```
hello;bonjour
cat;chat
dog;chien
```

Each word pair consists of an English word and its French translation separated by a semicolon.

3 Implementation

3.1 Version 1

- A simple server was developed to receive signals and select a random translation.
- The client program sent signals specifying the translation direction (English-to-French or French-to-English):

```
1     for (int i = 0; i < 100; i++) {
2         int signal = (rand() % 2) ? SIGUSR1 : SIGUSR2;
3         if (kill(server_pid, signal) == -1) {
4             perror("Failed to send signal");
5             return EXIT_FAILURE;
6         }
7         printf("Sent signal: %s\n", (signal == SIGUSR1) ? "
              SIGUSR1" : "SIGUSR2");
8         usleep(100000);
9     }
```

Listing 1: Client sending signals to the server

- The server monitored the dictionary folder and dynamically updated its dictionary based on file modifications:

```
    if (dir_stat.st_mtime != last_mtime) {
        printf("\n\nFolder␣modification␣detected!\n");
        load_dictionary(folder);
        printf("Dictionary␣updated!!\n\n");
        last_mtime = dir_stat.st_mtime; // Update the last
           modification time
    }
```

Listing 2: Dynamic dictionary update

3.2 Version 2

This version included the following enhancements:

- Use of shared memory for storing the dictionary.
- A writer program read word pairs from files and sent them to a message queue.

```
        if (english && french) {
            Message msg;
            msg.mtype = 1; // Message type
            strncpy(msg.english, english, 50);
            strncpy(msg.french, french, 50);
            msg.direction = 1; // English-to-French
            msgsnd(msgid, &msg, sizeof(Message) - sizeof(long
               ), 0);

            msg.mtype = 2; // Message type for French-to-
               English
            strncpy(msg.english, french, 50);
            strncpy(msg.french, english, 50);
            msg.direction = 2; // French-to-English
            msgsnd(msgid, &msg, sizeof(Message) - sizeof(long
               ), 0);

            printf("Sent:␣%s␣->␣%s\n", english, french);
        }
```

Listing 3: Writer program sending word pairs

- A reader program processed translations from the queue and updated the shared memory.

```
        while (1) {
            Message msg;
            if (msgrcv(msgid, &msg, sizeof(Message) - sizeof(
               long), 0, 0) != -1) {
                if (shared_word_count < MAX_WORDS) {
                    strncpy(shared_dict[shared_word_count].
                       english, msg.english, 50);
```

```
                    strncpy(shared_dict[shared_word_count].
                        french, msg.french, 50);
                    shared_word_count++;
                    printf("Stored: %s -> %s\n", msg.english,
                        msg.french);
                } else {
                    printf("Shared dictionary is full!\n");
                }
            }
        }
```

Listing 4: Reading message of client and updating shared memory

- Threading was implemented for concurrent processing.

```
        pthread_t reader_thread;
        pthread_create(&reader_thread, NULL,
            translation_reader_thread, &msgid);

        printf("Reader running and processing translations
            ...\n");
        pthread_join(reader_thread, NULL);
```

Listing 5: Threading

3.3 Version 3

This unified version combined the features of the previous versions. Client process creates a message of a word, which will be translated, and a signal type, indicating direction of the translation:

- Clients sent translation requests via a named pipe "/tmp/word_pipe", and responses were received through another pipe /tmp/word_response_pipe, as server and client processes were unrelated:

```
    while (1) {
        printf("Enter a word (or type '-1' to quit): ");
        scanf("%49s", word);

        if (strcmp(word, "-1") == 0) break;

        printf("Enter direction (1 for EN->FR, 2 for FR->EN):
            ");
        scanf("%d", &direction);

        send_request(word, direction == 1 ? SIGUSR1 : SIGUSR2
            );
        receive_response(); // Get the response from the
            server
    }
```

Listing 6: Client interaction with the server

- The server handled concurrent translation requests and folder monitoring using threads, protected by a mutex:

```
int main() {
    int shm_id;
    pthread_t client_thread, monitor_thread;

    shm_id = shmget(SHM_KEY, MAX_WORDS * sizeof(WordPair)
        , IPC_CREAT | 0666);
    dictionary = (WordPair *)shmat(shm_id, NULL, 0);

    printf("Server started. Waiting for client requests
        ...\n");

    pthread_create(&client_thread, NULL, client_handler,
        NULL);
    pthread_create(&monitor_thread, NULL, file_monitor,
        NULL);

    pthread_join(client_thread, NULL);
    pthread_join(monitor_thread, NULL);

    shmdt(dictionary);
    shmctl(shm_id, IPC_RMID, NULL);
    return 0;
}
```

Listing 7: Server initialization with threads

  - If a word was found, the server sent the translation to the client:

```
        for (int i = 0; i < word_count; i++) {
            if (signal == SIGUSR1 && strcmp(dictionary[i
                ].english, word) == 0) {
                snprintf(response, sizeof(response), "
                    Translation: %s -> %s", word,
                    dictionary[i].french);
                send_response(response);
                pthread_mutex_unlock(&mutex);
                return;
            }
            if (signal == SIGUSR2 && strcmp(dictionary[i
                ].french, word) == 0) {
                snprintf(response, sizeof(response), "
                    Translation: %s -> %s", word,
                    dictionary[i].english);
                send_response(response);
                pthread_mutex_unlock(&mutex);
                return;
            }
        }
```

Listing 8: Handling found word in shared memory

– If the word was not found, the dictionary was reloaded:

```
        snprintf(response, sizeof(response), "Word ’%s’ not found. Reloading dictionary...\n", word);
        send_response(response);

        load_dictionary();
        pthread_mutex_unlock(&mutex);
```

Listing 9: Handling word not found

4 Results

The project successfully achieved:

- Dynamic translation updates based on file modifications.
- Real-time translations for English-to-French and French-to-English requests.
- Efficient concurrent translation handling using threads.
- **Only in version 3** 3.3: Since this version was the most advanced one, the program includes threads, and mutexes. For example, while one of the threads in server side is working on the request coming from client side, the monitor thread can not update shared memory in case of a new file being introduced to the source folder, due to usage of mutex lock. This showcases how this version avoid race condition.

5 Analysis

5.1 Advantages of Using Two Pipes

- Smooth communication by separating data streams for requests and responses.
- Avoidance of deadlocks through independent operation for each direction.
- Simple implementation in UNIX/Linux environments.

5.2 Drawbacks of Two Pipes

- Increased hardware resource usage (buffers, file descriptors).
- Limited scalability for multi-client systems.
- Dependency on buffer limits, which may cause blocking under heavy loads.

6 Conclusion

This project demonstrated the practical application of IPC techniques to create a robust bilingual translation server. Each version progressively incorporated advanced features, showcasing the scalability and adaptability of the system.