

NAT NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

(KARACHI CAMPUS)

FAST School of Computing

**Spring 2024**

**PROJECT REPORT**





NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

(KARACHI CAMPUS)

FAST School of Computing

**Spring 2024**

**PROJECT:** Implementing Shared Files between two processes

**OBJECTIVE:**

The primary objective of this project is to demonstrate the implementation of a system where two processes can share a file using semaphores in C programming. The specific objectives include:

* Implementing semaphores to control access to the shared file.
* Developing file operations for reading and writing data to the shared file.
* Using shared memory for communication between the two processes.
* Synchronizing the processes using semaphores to prevent data corruption.

**PROJECT DESCRIPTION:**

This project implements a system in C programming where two processes can share a file using semaphores. Shared memory allows multiple processes to access the same memory segment, enabling them to share data efficiently. Semaphores are used to control access to the shared file, ensuring that only one process accesses the file at a time to prevent data corruption. In the mentioned tutorial, we'll create two programs: a client and a server. The client program will write data to shared memory, and the server program will read and display the data.

**CLIENT CODE:**

The client program is responsible for writing data to shared memory. It uses a semaphore to synchronize access to the shared memory segment. Here's the client code:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <fcntl.h>

#include <sys/stat.h>

#include <sys/mman.h>

#include <unistd.h>

#include <semaphore.h>

#define SEM\_NAME "/my\_sem"

int manipulateAndWriteToFile(char\* ptr, int SIZE, FILE\* file, sem\_t\* sem, int\* offset) {

char input[SIZE];

sem\_wait(sem);

printf("Enter data to write to shared memory (or type 'exit' to stop): ");

fgets(input, sizeof(input), stdin);

if (strcmp(input, "exit\n") == 0) {

sem\_post(sem);

return 1;

}

// Remove the newline character from the input string

size\_t input\_length = strlen(input);

if (input[input\_length - 1] == '\n') {

input[input\_length - 1] = '\0';

}

// Calculate the position to write the new data

int position = \*offset;

\*offset += snprintf(ptr + position, SIZE - position, "%s\n", input);

// Write to file

fprintf(file, "%s\n", input);

fflush(file); // Flush the output buffer to ensure data is written immediately

sem\_post(sem);

printf("Data written to shared memory and updated in file successfully.\n");

return 0;

}

int main() {

const char\* file\_name = "example.txt";

const char\* name = "OS";

const int SIZE = 4096;

int shm\_fd;

char\* ptr;

sem\_t\* sem;

// Open semaphore

sem = sem\_open(SEM\_NAME, O\_CREAT, 0666, 1);

if (sem == SEM\_FAILED) {

perror("Semaphore initialization failed");

return 1;

}

// Open shared memory

shm\_fd = shm\_open(name, O\_CREAT | O\_RDWR, 0666);

if (shm\_fd == -1) {

perror("Shared memory failed");

return 1;

}

// Resize shared memory

ftruncate(shm\_fd, SIZE);

// Map shared memory

ptr = mmap(0, SIZE, PROT\_READ | PROT\_WRITE, MAP\_SHARED, shm\_fd, 0);

if (ptr == MAP\_FAILED) {

perror("Map failed");

return 1;

}

// Create or truncate file

FILE\* file = fopen(file\_name, "w");

if (file == NULL) {

perror("Error opening file");

exit(1);

}

fclose(file);

// Open file for appending

file = fopen(file\_name, "a");

if (file == NULL) {

perror("Error opening file");

exit(1);

}

// Loop for writing to shared memory and file

int num\_ofChar = 0;

while (1) {

if (manipulateAndWriteToFile(ptr, SIZE, file, sem, &num\_ofChar)) {

break;

}

}

// Close the file

fclose(file);

// Read and print content of shared memory

printf("Shared Memory Content:\n");

printf("%s\n", ptr);

// Unmap shared memory

if (munmap(ptr, SIZE) == -1) {

perror("munmap");

return 1;

}

// Close shared memory

close(shm\_fd);

// Close semaphore

sem\_close(sem);

return 0;

}

**SERVER CODE:**

The server program reads data from shared memory and displays it. It also uses a semaphore for synchronization. Here's the server code:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <fcntl.h>

#include <sys/stat.h>

#include <sys/mman.h>

#include <unistd.h>

#include <semaphore.h>

#include <errno.h>

#define SEM\_NAME "/my\_sem"

void readSharedMemory(const char\* ptr, int size) {

// Copy shared memory data to a separate buffer

char\* buffer = malloc(size + 1); // Allocate memory for buffer (+1 for null terminator)

if (buffer == NULL) {

perror("Memory allocation failed");

return;

}

memcpy(buffer, ptr, size); // Copy shared memory data to buffer

buffer[size] = '\0'; // Null-terminate the buffer

printf("Data read from shared memory:\n");

char\* line = buffer;

char\* token;

while ((token = strsep(&line, "\n")) != NULL) {

printf("%s\n", token);

}

free(buffer); // Free allocated memory

}

int main() {

const char\* name = "OS";

int shm\_fd;

char\* ptr;

sem\_t\* sem;

int read\_flag = 0; // Flag to track if memory has been read

sem = sem\_open(SEM\_NAME, O\_CREAT, 0666, 1);

if (sem == SEM\_FAILED) {

perror("Semaphore initialization failed");

return 1;

}

while (!read\_flag) {

shm\_fd = shm\_open(name, O\_RDWR, 0666);

if (shm\_fd == -1) {

perror("Shared memory failed");

return 1;

}

struct stat s;

if (fstat(shm\_fd, &s) == -1) {

perror("fstat");

return 1;

}

int size = s.st\_size;

ptr = mmap(0, size, PROT\_READ, MAP\_SHARED, shm\_fd, 0);

if (ptr == MAP\_FAILED) {

perror("Map failed");

return 1;

}

sem\_wait(sem);

readSharedMemory(ptr, size);

sem\_post(sem);

if (munmap(ptr, size) == -1) {

perror("munmap");

return 1;

}

if (close(shm\_fd) == -1) {

perror("close");

return 1;

}

read\_flag = 1; // Set flag to indicate memory has been read

}

if (sem\_close(sem) == -1) {

perror("sem\_close");

return 1;

}

if (sem\_unlink(SEM\_NAME) == -1) {

perror("sem\_unlink");

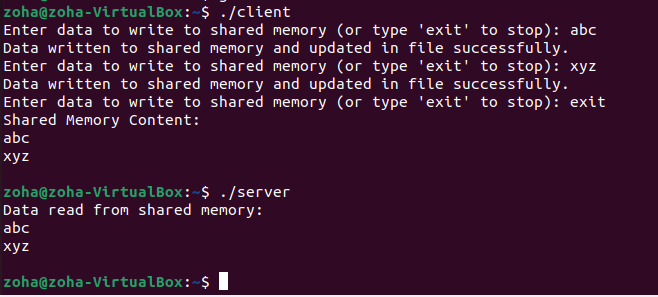
return 1;

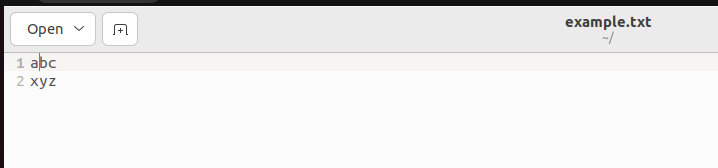
}

return 0;

}

**SCREENSHOTS:**





**TUTORIAL:**

**Step 01: Include Necessary Libraries**

Both the client and server programs require several standard C libraries for file handling, memory mapping, and semaphore operations. Include these libraries at the beginning of your source files:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <fcntl.h>

#include <sys/stat.h>

#include <sys/mman.h>

#include <unistd.h>

#include <semaphore.h>

**Step 02: Define Shared Memory and Semaphore Names**

Define constants for the shared memory name and semaphore name. These names must be unique and should be the same in both the client and server programs:

#define SEM\_NAME "/my\_sem"

#define SHM\_NAME "OS"

#define SHM\_SIZE 4096

**Step 03: Implement Functions for Client and Server**

**Client: Writing to Shared Memory**

The client program needs a function to write data to shared memory. The function should take the shared memory pointer, its size, a file pointer for logging, the semaphore for synchronization, and the current offset as arguments. It should return 1 to exit or 0 to continue writing:

int manipulateAndWriteToFile(char\* ptr, int SIZE, FILE\* file, sem\_t\* sem, int\* offset);

**Server: Reading From Shared Memory**

The server program needs a function to read data from shared memory. This function should take the shared memory pointer and its size as arguments and should print the data:

void readSharedMemory(const char\* ptr, int size);

**Step 04: Implement Functions for Client and Server**

**Client Main Function:**

In the client's **main** function, open the semaphore, shared memory, and the file for logging. Then, enter a loop to write data to shared memory and the file:

int main() {

// Open semaphore, shared memory, and file

// Loop for writing data to shared memory and file

// Close file, shared memory, semaphore, and return

}

**Server Main Function:**

In the server's **main** function, open the semaphore, and continuously read data from shared memory until the client signals completion:

int main() {

// Open semaphore

// Loop for reading data from shared memory

// Close semaphore and return

}

**Step 05: Compile and Run**

Compile the client and server programs separately using the following commands:

gcc client.c - o client

gcc server.c - o server

Run the server program first, followed by the client program. You should see the data written by the client displayed by the server.

**CONCLUSION:**

In conclusion, we have demonstrated how semaphores can be used to implement shared files between two processes in C programming. Semaphores provide a simple and efficient way to control access to shared resources, ensuring that only one process accesses the file at a time.

**PROJECT TEAM:**

* Muhammad Shayan Shafique (20K-1899)