

VietNam National University Ho Chi Minh City
Ho Chi Minh City University of Technology
Faculty of Computer Science and Engineering



Operating System Course Laboratory 2 Report

CC07

Student Name	Student ID
Phan Ba Thanh	2353084

March, 2025

1 Problem 4.1

Using shared memory to compute average movie ratings from input files using two child processes.

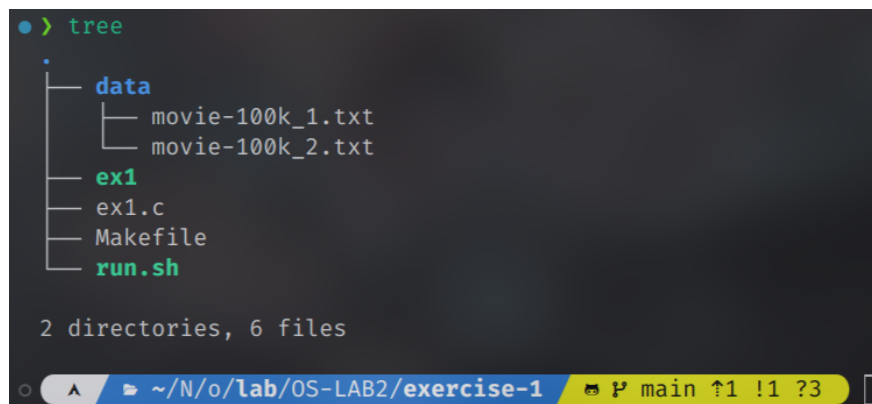
The program reads movie ratings from a file, computes average ratings, and stores results in shared memory. The key components include:

- Shared memory for interprocess communication (IPC).
- Forking two child processes to handle separate files.
- Computation of average ratings per movie.
- Synchronization using wait mechanisms.

1.1 Implementation Details

1.1.1 Structure

The two given txt file of the movies information is stored in exercise-1/data. The main, Makefile, run.sh is in the folder exercise-1.



```
> tree
.
├── data
│   ├── movie-100k_1.txt
│   └── movie-100k_2.txt
├── ex1
├── ex1.c
├── Makefile
└── run.sh

2 directories, 6 files
```

Figure 1: Exercise organizing.

1.1.2 Shared Memory Setup

The program uses the System V shared memory API:

```
int shmid = shmget(IPC_PRIVATE, size, IPC_CREAT | 0666);
double *shm_ptr = shmat(shmid, NULL, 0);
```

This allocates a shared memory segment and attaches it to the process address space.

1.1.3 Child Process Execution

Each child process reads a different file, computes averages, and writes to shared memory:

```
pid_t pid = fork();
if (pid == 0) {
    compute_average(filename, shm_ptr, offset);
    exit(0);
}
```

The parent waits for both children to complete before printing results.

1.1.4 Computing Averages

The `compute_average()` function processes input as follows:

- Reads ratings from the file.
- Maintains cumulative sums and counts.
- Computes averages and stores them in shared memory.

1.2 Run the program

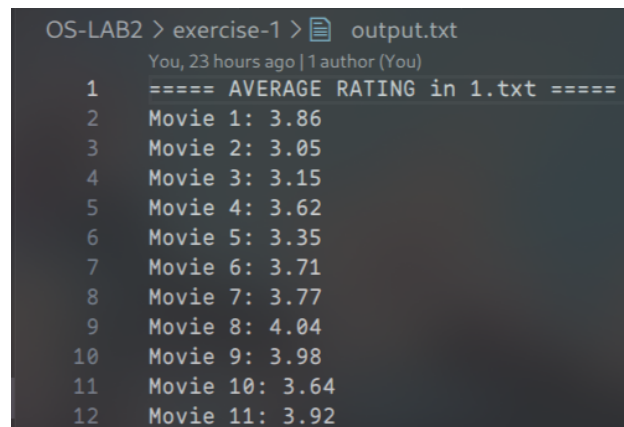
The program is compiled with:

```
make
```

It runs using:

```
./ex1
```

The computed average ratings are printed and stored in `output.txt`.



```
OS-LAB2 > exercise-1 > output.txt
You, 23 hours ago | 1 author (You)
1  ===== AVERAGE RATING in 1.txt =====
2  Movie 1: 3.86
3  Movie 2: 3.05
4  Movie 3: 3.15
5  Movie 4: 3.62
6  Movie 5: 3.35
7  Movie 6: 3.71
8  Movie 7: 3.77
9  Movie 8: 4.04
10 Movie 9: 3.98
11 Movie 10: 3.64
12 Movie 11: 3.92
```

Figure 2: Output file.

2 Problem 4.2

Write a program to solve Problem 1 using a multi-threaded process.

The program uses POSIX threads (pthreads) to calculate average movie ratings from input files. The program ensures thread safety using mutex locks¹ and executes in parallel to improve efficiency. The program reads movie ratings from a file, computes average ratings, and stores results in global arrays protected by a mutex. The key components include:

- POSIX threads for parallel computation.
- Mutex locks for synchronization.
- Computation of average ratings per movie.

¹About the mutual exclusion definition, I have looked it up in the book Operating System Concepts 10th edition to learn how to implement it because it is not mentioned in the lab specs

2.1 Implementation Details

2.1.1 Structure

This exercise folder organizing is quite similar to the first one. However, this time I calculate the average of the movies ratings in both files and merge it into one section of output.

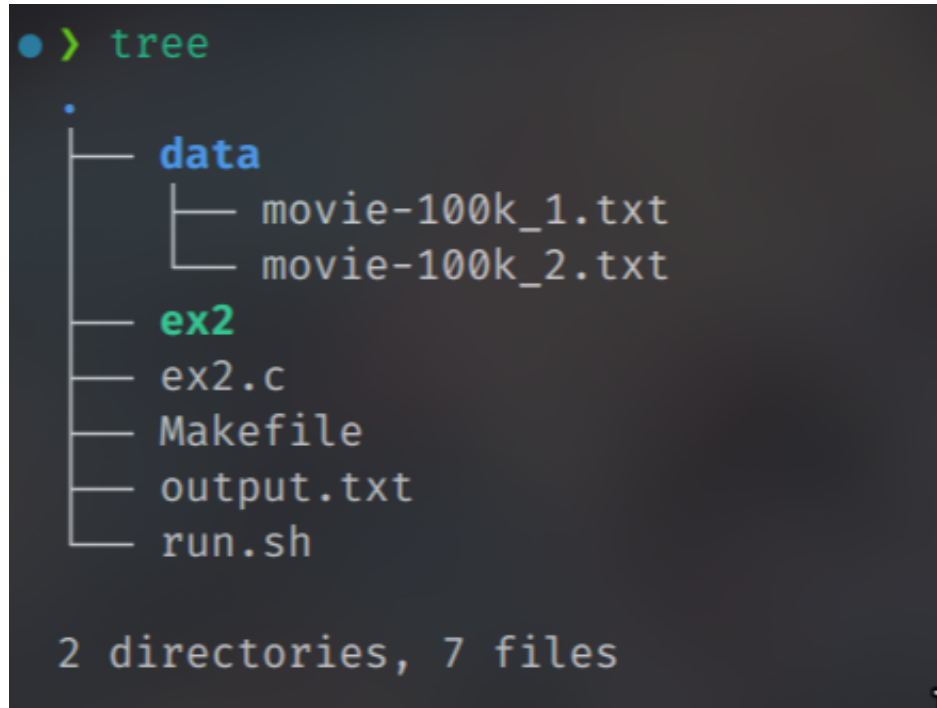


Figure 3: Problem structure.

2.1.2 Thread Creation and Execution

Threads are created for each file using the `pthread_create()` function:

```
pthread_t tid;
pthread_create(&tid, NULL, compute_average, (void *)filename);
```

Each thread executes `compute_average()` independently.

2.1.3 Computing Averages with Synchronization

Although creating global variables are considered bad practice, I stucked with the idea of creating two arrays of sum and count. Since they are global vars, both threads can be manipulated successfully.

The `compute_average()` function processes input as follows:

- Reads ratings from a file.
- Uses a mutex lock to ensure safe updates to shared data.
- Computes and stores averages in global arrays.

The mutex lock prevents race conditions:

```
pthread_mutex_lock(&lock);  
sum_ratings[movieID] += rating;  
count_ratings[movieID]++;  
pthread_mutex_unlock(&lock);
```

2.2 Run the program

The program is compiled with:

```
make
```

It runs using:

```
./ex2
```

The computed average ratings are printed and stored in output.txt.

3 Problem 4.3

The program uses FIFO (named pipes) for interprocess communication (IPC). The program enables processes to send and receive messages through a named pipe. It creates a FIFO file and uses two functions for communication:

- 'fifo_snd()': Writes messages to the FIFO.
- 'rcv()': Reads messages from the FIFO.

3.1 Implementation Details

3.1.1 Structure

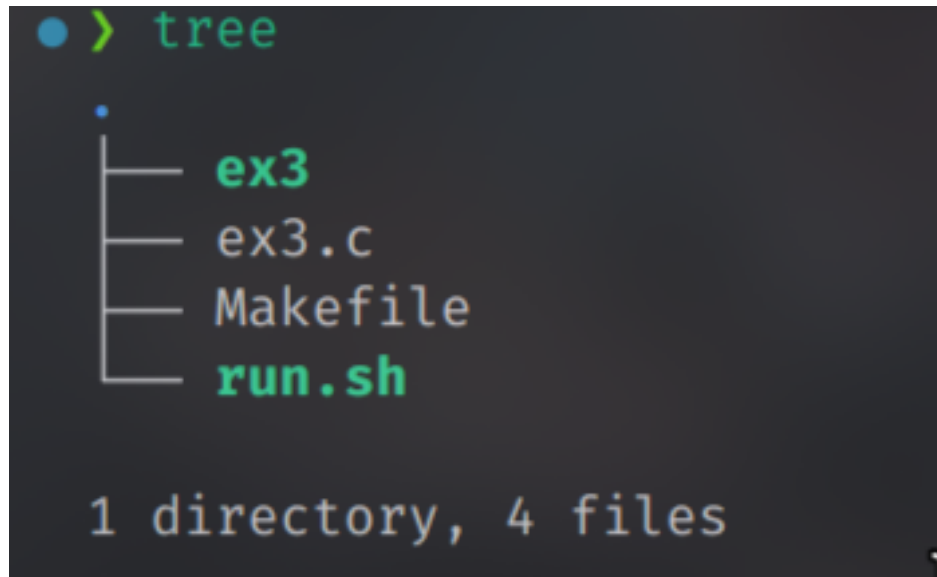


Figure 4: Problem structure.

3.1.2 FIFO Creation and Setup

The FIFO file is created using:

```
mkfifo("abc_fifo_name", 0666);
```

This creates a named pipe with read/write permissions.

3.1.3 Writing to the FIFO

The 'fifo_snd()' function writes messages:

```
int fd = open(FIFO_FILE, O_WRONLY);
write(fd, message, strlen(message) + 1);
close(fd);
```

This opens the FIFO in write mode and sends a message.

3.1.4 Reading from the FIFO

The 'rcv()' function reads messages:

```
int fd = open(FIFO_FILE, O_RDONLY);
read(fd, buffer, sizeof(buffer));
printf("Message received haha: %s\n", buffer);
close(fd);
```

This opens the FIFO in read mode and prints the received message.

3.2 Run the program

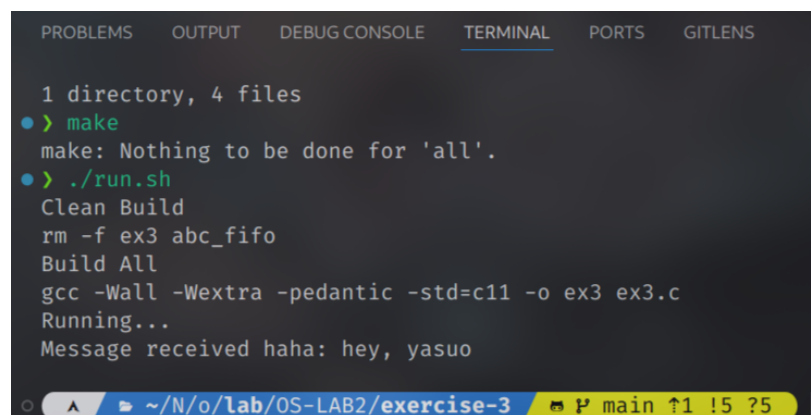
The program is compiled with:

```
make
```

It runs using:

```
./ex3
```

The sender writes a message, and the receiver prints it.



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS

1 directory, 4 files
● > make
make: Nothing to be done for 'all'.
● > ./run.sh
Clean Build
rm -f ex3 abc_fifo
Build All
gcc -Wall -Wextra -pedantic -std=c11 -o ex3 ex3.c
Running...
Message received haha: hey, yasuo

~ / N / o / lab / OS - LAB2 / exercise - 3  P main ↑1 !5 ?5
```

Figure 5: Result on console.

4 Problem 4.4

I have not finished.

5 Problem 4.5

The program in this exercise creates a file (`shared_memory.txt`), truncates it to a specific size, and then maps that file into the address space of the parent and child processes using `mmap`. This directly fulfills the requirement of "mapping a created file into local address space." The parent process writes a message to the mapped memory, and the child process reads it, demonstrating data sharing.

5.1 Implementation Details

The C code performs the following key steps:

1. File Creation and Sizing:

- The `create_file` function is responsible for creating a file named `shared_memory.txt`.
- `open` system call is used to create/open the file with read/write permissions.
- `ftruncate` is then used to resize the file to a predefined size (`FILESIZE`), which will be the size of the shared memory region.

2. Memory Mapping:

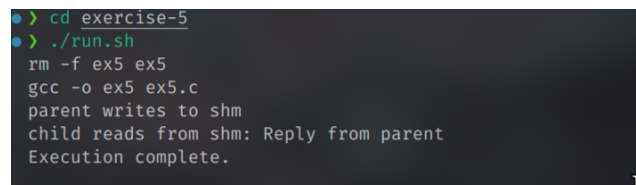
- The `mmap` system call maps the created file into the address space of the process.
- `PROT_READ` and `PROT_WRITE` flags allow both read and write operations on the mapped memory.
- `MAP_SHARED` specifies that changes made to the mapped memory are visible to other processes that map the same file.

3. Process Creation and Communication:

- `fork` creates a child process.
- The parent process writes a message into the shared memory region.
- The child process reads the message from the shared memory region.
- `sleep` is used in the child process to ensure the parent writes the data first (this is a simple synchronization mechanism).

5.2 Run the program

Using `./run.sh`.



```
➤ > cd exercise-5
➤ > ./run.sh
rm -f ex5 ex5
gcc -o ex5 ex5.c
parent writes to shm
child reads from shm: Reply from parent
Execution complete.
```

Figure 6: Results