OPERATING SYSTEMS

Assignment 3

Pavlos Ntais

Contents

1 General Information

The assignment has been implemented in C. The functionalities described in the assignment, have been implemented. I have also thoroughly tested the program, and it works as intended.

2 Directory & File Explanation

I have chosen the following file organization:
src/creator.c: Responsible for creating shared memory and managing readers & writers.
src/utilities.c: General utility functions used in the programs.
src/Reader/reader.c: Implementation of the reader.
src/Reader/reader_utilities.c: Helper functions for the reader.
src/Writer/write.c: Implementation of the writer.
src/Writer/write_utilities.c: Helper functions for the writer.
include/: Header files for the program.

3 Synchronization Mechanism

3.1 General Idea

For synchronization purposes, we treat each region that either the reader reads or the writer writes to as its own critical section. Therefore, there can be multiple critical sections simultaneously in the file.

The idea is simple: we have two arrays of fixed size, one for active readers and one for active writers, as well as a region where they interact. This is how we define critical sections. This way, only a predetermined number of processes can control the critical section at any given time. Specifically, the idea is as follows:

When a process **enters**, it checks if it interferes with another processes' region (meaning critical section) and keeps the count of blocked regions/processes (the variable **blocked_by**). If it's more than one, it performs a P operation on the semaphore, resulting in it being suspended on that semaphore.

When a process **finishes** its execution, it checks if it interferes with another process's region that entered after it. If so, it decrements, as it is no longer blocked. If it becomes 0 (blocked_by), it means it is not blocked by anyone. Therefore, it performs a V operation on the semaphore, allowing it to take control. Essentially, this creates a First In First Out (FIFO) system, strictly following the order of entry for both readers and writers, avoiding process starvation.

3.2 Necessary Conditions

It is important to note that with this technique, we ensure the fulfillment of the three synchronization rules.
Mutual Exclusion: Two processes cannot be in the same critical section simultaneously, as before entering, a process individually checks if someone else is already there. If so, it will only enter its CS when everyone ahead of it finishes (FIFO). Thus, we fulfill the mutual exclusion principle.

Progress: With the exit of a process from the critical section and entry into the remaining section, it does not participate in the decision of whether it will re-enter the CS. Also, the decision about who will be the next to enter the critical section, as explained below in detail, is not postponed indefinitely. Therefore, we have continuous progress.

Bounded Waiting: There is a limit to the number of times a process (either a reader or a writer) will wait. Specifically, each process needs to wait for a predetermined (bounded) number of processes that block it. This number is, at worst, the size of the fixed array when initially blocked by all. However, it is given that, with the end of the processes ahead of us, at some point, we will take control of the program. Thus, we have bounded waiting by definition.

4 Creator

The creator is responsible for creating shared memory, initializing it, and forking + executing readers/writers. More specifically, the following steps are followed:

1. Read the command line arguments.
2. Create the shared memory segment and the semaphores (further explained in technical details).
3. Deploy the readers and writers with random parameters.
4. When the processes finish, display the statistical information from the execution.
5. Unlink the shared memory and destroy the semaphores.

5 Reader

For the reader, we follow these steps:

1. Read the command line arguments.
2. Open and load the shared memory segment.
3. Wait to enter the array of processes (we have a bounded number of active processes).
4. Find the writers writing within our range (using the writer's array) and store their number. If there is at least one, a P operation is performed on its semaphore and it waits to be unblocked upon the completion of the respective writer(s).
5. Open the file and read the records. The records are first stored in an array to avoid constant read system calls, which significantly slow down the program for a large number of records.
6. After reading, find the writer(s) we used to block and decrease the number of readers blocking them (blocked_by). If it becomes 0, perform a V operation on the semaphore and unblock the writer.

7. Atomically update the statistics, close the file, free the allocated memory, and exit the program.

6 Writer

For the writer, we follow these steps:

1. Read the command line arguments.
2. Open and load the shared memory segment.
3. Wait to get a turn in the array of processes (with a bounded number of active processes).
4. Find the readers/writers within the writer's range who are reading/writing (via the array) and store their numbers. If there is at least one, perform a P operation on our semaphore and wait to be unblocked upon the completion of these processes.
5. Open the file and update the record.
6. After writing, find the readers/writers we block and decrease the count of processes blocking them. If it becomes 0, perform a P operation on the semaphore and give control back to the process.
7. Update the statistics individually, close the file, and exit the program.

7 Logger

To ensure the correctness of the synchronization, a logging system has been implemented, printing information about the entry and exit of readers and writers to a file. For logging, there are 3 (three) information levels to allow the user to choose the desired level of abstraction. The file where the logging occurs is **logging_info.log**. To choose the logging level, the user needs to pass the number of the desired level (1/2/3).

7.1 Simple Logging

Prints the entry and exit of readers/writers to the critical section. A total of 2*(+) prints.

7.2 Medium Logging

Provides a lower-level abstraction and prints the specific process entered in the array of processes, as well as the number of processes from which it is blocked, if such processes exist. Prints the entry and exit of readers/writers to the critical section. total of 3*(+) prints.

7.3 Advanced Logging

The third, and final level provides a complete view of the synchronization process. We print everything provided in Level 2, as well as the **specific** processes from which we are blocked (upon entry) and the processes we block (upon exit). Also, prints the active readers & writers (those with the ability to enter the CS) in an array format. The number of prints is likely much higher than the second level.

8 Technical Details

1. For the implementation of shared memory, the POSIX API was preferred. The POSIX API was also chosen for semaphores. Further information is referred to in the sources.

2. To maintain the insertion order, crucial for waking up the processes we block (i.e., processes that entered after us and "hit" our range), a ticket system is implemented. When a reader/writer enters the array of active processes, it takes a ticket indicating the order in which it entered. So, when a process ends, we only need to look at processes with a higher ticket than ours, which means they entered later on.

3. Readers/writers sleep after reading/writing for a random time between 1 and the given time. This is done to simulate the time needed in a real scenario.

4. The process of reading is as follows: a) We transfer the records from the file to a temporary array. This is done because constant reads from the file make the program (as I discovered while testing) up to 3 times slower due to the consistent system calls. b) Individual readers then print the records to stdout for a uniform appearance.

9 Usage

Execution is done using the command

./bin/creator -f **file** -rn **readers_number** -wn **writers_number** -rt **readers_time** -wt **writers_time** -wp **writer_exec** -rp **reader_exec** -sm **shared_memory_name** -lg **log_level**

However, the use of a makefile is provided with the following parameters:
make - Compilation and linking of the program.
make run - Execution of the program (with default command line arguments).
make file - Execution of the program and passage of results to an output.txt file (with default command line arguments).
make help - Execution of the program using valgrind.
make clear - Deletion of object files and the executable created during compilation.
make clear_all - Deletion of object files, the executable created during compilation, the shared memory segment, and the logging file.