Operating Systems – Project Programming #3 – System Design Writeup

Group 9:

Alex Ko - fyk211

Ritin Malhotra - rm5486

Alex and Ritin have created a system that allows different readers and writers to concurrently access and modify records from a binary data file, ensuring that the processes wait for each other to finish appropriately and do not starve.

The Structure of the Program:

Our program comprises the following:

1. A **studentRecord** struct that comprises the following data:

```
#define NAME_LENGTH 20 // Max length of a name
#define NUM_COURSES 8 // Max number of courses for each student
typedef struct
{
 long studentID; // Unique ID of a student
 char lastName[NAME_LENGTH]; // Last name of student
 char firstName[NAME_LENGTH]; // First name of student
 float grades[NUM_COURSES]; // Maximum number of courses
 float GPA; // GPA of student
} studentRecord;
```

   This struct contains the data for each student, which we can look up and edit using readers and writers. This struct resides in the “common.h” file

2. **Coordinator**: A coordinator which loads up an array containing the IDs of the students in a shared memory segment. We load the IDs of each student instead of the entire record since we were running into segmentation faults while loading the entire set of records,

primarily because the coordinator could not upload all the records to the shared memory for reasons that are currently unknown to us. However, we are still able to access the entire record of the student given their ID by finding it in the binary data file through the following formula - index of ID in shared array * size of studentRecord struct. This enables us to quickly access the records, while ensuring that the shared memory is properly able to hold the information for all students.

The coordinator is also responsible for getting and displaying the following statistics regarding the execution of the program:

- number of readers processed - attained via a semaphore that updates upon completion of a reader process
- average duration of readers - attained via a double variable shared through a shared memory segment
- number of writers processed - attained via a semaphore that updates upon completion of a writer process
- average duration of writers - attained via a double variable shared through a shared memory segment
- maximum delay recorded for starting the work of either a reader or a writer - attained via a double variable shared through a shared memory segment
- sum number of records either accessed or modified - attained via two semaphores

At the end of its execution, the coordinator purges the shared memory segments and the semaphores it uses to collect the statistics pertaining to program execution.

3. **Helper Functions:** Before moving on to the readers and writers, we would like to look at a few **helper functions** that enable the readers and writers to properly access student

records from the shared memory segment containing student IDs and the binary data files passed as arguments:

1. The **findRecord** function that takes in a studentID, the index of the studentID in the shared array of studentIDs and the array of studentIDs. This function finds the student's information from the binary data file given their studentIDs and the index of the studentID in the shared array of studentIDs using the following formula:
Location of student information in BIN file = index of studentID in shared array * size of studentRecord struct.
The function returns a studentRecord struct with the appropriate information if it finds it. In case it does not, it returns a NULL value. This function is useful since it enables us to swiftly find data about a student from the binary data file given their studentID.
2. The **editRecord** function that takes in a studentID, the index of the studentID in the shared array of studentIDs and the array of studentIDs. This function finds the student's information and stores it in a studentRecord struct pointer using the findRecord() function. It then enables the writer to edit any grade of the student. However, we have implemented functions that ensure that the writer writes to the record within the time it has left to access the record, as determined by the time argument passed to it and the time at which this function is called. This is important as it ensures that a writer **does not starve** any other process by infinitely staying in a blocking I/O operation like scanf(). This function ensures that the writer is able to record, and more importantly, that it does so within the time that it has left.

4. **Reader**: A **reader** which reads information about each student from the shared memory segment and the binary data file containing the student's records. This program first loops

through all the record IDs that the user wants to check, and then forks a child process for each ID. This approach ensures that we can concurrently search for each record ID and print out each record. If one of the records is currently being written, or the semaphore for it is being held, then this reader will wait until all the child processes are done with their progress. Inside a child, it first checks the presence of a writer by using named semaphores. If a writer is currently modifying the same record, then the reader waits. If not, it proceeds to find the record and then prints out the information about a particular student. Currently, the maximum number of readers is set to 2 to avoid having an infinite number of readers as well as semaphores in the system per record. We have made this choice as per the advice of our mentors for the course. If there are more than the maximum number of readers wanting to read right now, only two are granted access to read while others have to wait for them to release the semaphores. Every time a reader finishes its job, we use the sem_getvalue() function to constantly check if there is any reader left. If there is none, we destroy the read semaphore for this record, so that we free up the system resources and let a pending writer go through.

5. **Writer**: A writer which updates a studentRecord associated with a studentID it is provided via its args. Similar to the reader, a writer also need to check whether there is a reader present. If there is one, the writer has to wait until all readers finish their jobs. If there is none, then the writer can proceed to write to the record specified by the user. There can be only one writer at one time per record, which is different from readers. The writer takes an input from the user and checks which course grade they would like to modify, and recalculates the gpa for this particular student.

6. **Time:** As specified by the assignment requirements, each process should "work" for a specified amount of time before exiting execution. As such, we have implemented a system where a process either exits when it is done working for a specified amount of time, or "sleeps" to simulate doing work for the time left in its execution. This way, we can avoid starvation by ensuring that the processes exit after a certain amount of time, regardless of whether they've completed their work or not.
7. **Purging**: Our program frees up all the semaphores and shared memory from the system resources once our program terminates. A status message will be shown at the end of each coordinator, writer, and reader's execution to display if all the semaphores and shared memory have been purged successfully. We also handle the case where a program gets terminated by control+c accidentally. We registered a cleanup handler that cleans up all the used resources every time the program gets terminated by control+c. This is a careful way to handle such edge cases and make sure that system resources don't get depleted.
8. **Starvation**: our solution to tackle the program of starvation is that we won't allow any reader or writer to do its job indefinitely. Each reader and writer exits and purges all the semaphores after the time specified by the user. This way we can ensure that all the readers and writers have a chance to eventually access the shared resources and won't have to wait indefinitely.

Running the Application:

1. Firstly, please “make clean” and “make” (without quotes) to ensure that the output files are up-to-date and properly able to run on your system. Our testing on the linux remote server shows no errors on this step.
2. Then, run the coordinator by running ./coordinator -f filename -s shmid. This will initialize the shared memory segments as well as the semaphores used to generate the stats at the end of program execution.
3. Then, run a reader or writer process. They take in the following arguments:
    - An -f flag to denote the binary data file containing student records. We have primarily used Dataset-500.bin while testing our program
    - An -l flag to denote the following:
        a. A list of record IDs (containing at least one record) if you’re running a reader process
        b. One recordID if you’re running a writer process
    - A -d flag to denote the time each process runs for (in seconds)
    - An -s flag to denote the key of shared memory segment containing the array of studentIDs. Kindly use the same key passed as an argument to the coordinator to avoid errors. We recommend using 100 as the key (since it’s a nice number that made testing easier).
    - Here are some examples:

      To run the coordinator: ./coordinator -f Dataset-500.bin -s 100

      To run a writer: ./writer -f Dataset-500.bin -l 100007 -d 10 -s 100

      To run a reader: ./reader -f Dataset-500.bin -l 100007 -d 10 -s 100

To read multiple records: /reader -f Dataset-500.bin -l 100007,101344 -d 10 -s 100

4. Once the processes are done executing, kindly end the program's execution by giving the coordinator an "**exit**" command (kindly avoid using ctrl-C here). This will detach all shared memory segments and unlink the semaphores used for data collection. The coordinator will print out the post-execution statistics discussed above and end the program gracefully.