Preamble:

In this assignment, you will write independent programs that ***run concurrently*** and either *read (access)* or *write (update)* student records stored in a **binary data-file**. These programs can be used by professors and university administrators at the end of each academic year to initially spot and subsequently correct marks that have been incorrectly recorded. The simultaneous operation of the above programs constitutes the ***readers***/*writers* **problem**.

A binary data-file stores student records for a single year. Each such record consists of:

1. ***studentID***: a 8-character unique identifier,
2. ***student last name***: up to 20 character string,
3. ***student first name***: up to 20 character string,
4. ***grades for courses***: every student can take up to 8 maximum number of courses per year. Each such course receives a mark in accordance to the following grading scheme: 4.00 (for A), 3.50 (for A-), 3.00 (for B+), 2.50 (for B), 2.00 (for B-), 1.50 (for C+), 1.00 (for C), 0.50 (for C-). For courses either not taken or simply failed, the mark is set to 0.00.
5. ***GPA***: the grade point average of the student for the year.

The **access protocol** for reading/modifying student records is rather simple: you may have one or more readers at the same time looking up records from the binary data file. If a writer shows up with the intention of changing a (random) mark of a student and recalculating the corresponding GPA, it has to wait until *all* conflicting readers complete their work on the record and exit before the writer in question proceeds. Once a writer obtains the right of the way, it goes ahead and changes the content of designated records (i.e., modifies some or all the grades and recalculates the GPA). While a writer is at work, no other writer or reader can access the specific record ***undergoing modification*** from the binary data file. Every writer works with a **single record at a time**.

Readers and writers start their work at different and distinct points in time and their operations may randomly vary in length. For instance, a reader might "stay" and work with some records for 100 seconds while a writer may only desire to "work with a specific student record" and update its content for only 5 seconds. Such time-delays are *user-specified*.

Using the above protocol and depending on how we develop the program skeleton for readers, we may starve the writers. The overall objective of this project is to provide a solution that is *starvation-free* for both readers and writers.

You will have to demonstrate the correctness of your solution by launching different programs possibly from different `ttys` or by `fork()`-ing different readers/writes processes.

In the context of this programming assignment, you will:

- introduce a *shared memory segment* featuring objects that enable the operational protocol as well as a set of semaphores to facilitate the starvation-free operation.
- have all reader/writer processes attach the above shared segment so that they can concurrently access the content of the binary text-file of records for user-specified periods of time.
- use *(POSIX) Semaphores* to implement a *starvation-free* protocol and have readers and writers coordinate their work using appropriate P() and V() synchronization primitives.
- provide the capability for readers/writers to **stay working with records** for (varying) user-specified periods of time.

- have writers change the content of individual records. Readers can subsequently lookup changed content.

Project Description:

Figure 1 depicts how matters unfold when diverse operations arrive and work off a file made up of 22 distinct student records over time. Each reader may access one or multiple records. A write modifies one record at a time. Records for accessed are randomly selected. Readers and writers arrive at different times and once they get access of records of their interest, they "stay on" there working for a time period provided in the command line of the respective programs.

In Figure 1, readers and writers arrive one at time in the following chronological order: *R0, R1, R3, W0, R2, R3, R4, W1, W2, R5* and *W3*. Reader *R2* arrives at time 31 and looks up records 3, 4 and 7, while writer *W0* arrives at time 23 and intends to update record 4.

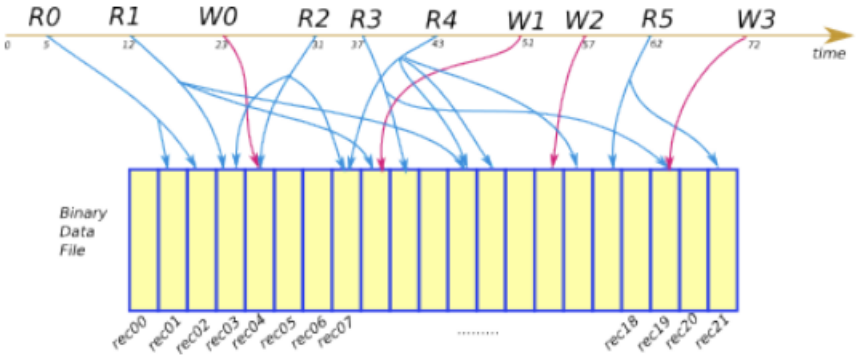

Figure 1: Multiple concurrent Readers/Writers working off a common binary data-file of student records

While working with the records, readers and writers must comply with the following restrictions at all times:

- One or multiple readers can simultaneously read the content of records provided that no writer is present.
- Only one writer at a time can update a record from the data file. Different writers may concurrently update different records.
- Once a writer starts its work, neither readers nor other writers are permitted to access the specific record in the data-file. Pending processes (of any type), must wait until the writer in question completes its work and exits.
- No readers/writers should suffer from *starvation*.
- Each writer updates a record of its choice modifying all or some of the grades recorded per record and recalculates the corresponding GPAs. Once the flushing of record(s) to the data file occurs, a writer waits by `sleep`-ing as many seconds as its command line specifies before it finally concludes its work.

- A reader looks up the content of a number of records, waits by `sleep`-ing for as many seconds as its command line designates and subsequently prints the record(s) out to its standard output before it ultimately terminates.

In your implementation, you should introduce a **shared-memory segment** where you can maintain structures and/or variables accessible by all your concurrent programs. For instance, you could deploy arrays of pids that would provide the process-ids of all active readers and writers at any single time instance. In this area, the values of additional objects could be held that ultimately help compile statistics regarding the operation of all your programs. Figure 2(a) depicts the shared memory elements that could help record the state-of-affairs just as the work of *R1*, *R2* and *W1* is underway. The fixed-size arrays should feature the

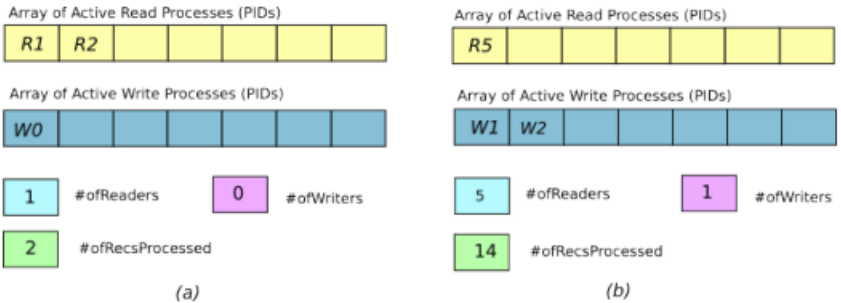

Figure 2: An example of shared-memory resident objects that could help in both coordination and compilation of statistics for *concurrent readers & writers*.

pids of processes active in the data file at say moment 35. Here, the first array indicates 2 active processes performing look-ups and 1 writer carrying out a modification. So far, 2 record-accesses/modifications have occurred. Similarly, Figure 2(b) shows how the content of structures that help oversee the operation in shared memory, has changed as the work of writers $W1$, $W2$ and the reader $R5$ are underway. Accrued statistics have changed accordingly.

Designing your Programs:

You are free to adopt any structure you wish for your reader/writer programs. Also you may introduce a (limited) number of **semaphores**, auxiliary structures and possibly other programs/executables. For instance, it would be a good idea to develop a program that functions as ***coordinator*** and creates up-front a shared segment, initializes it to appropriate objects and finally, makes the ID of this shared segment known to readers/writers. This coordinator program could also introduce the required semaphores as well as any auxiliary data structures you have to put in place in order to achieve a starvation-free solution. Potentially, the coordinator might **display** the state of the shared memory to help monitor the development of the various operations on the data file.

At the end of the execution of all readers/writers, you should always ensure that a process –perhaps the coordinator– that *cleans up* and *purges* both memory segment and semaphores so that system resources get properly released. If this clean up does not occur, **system pertinent resources may be depleted** and it will **ultimately become unable to claim additional segments.**

Every program should report its initiation, and termination time including artificial delays (or sleeps), as well as the time that it got hold of the records it worked on. Your programs should also create a **log** that is *readily understood* and can help in verifying the correctness of the operations of all processes involved.

You can invoke your programs from either different `ttys` by typing in appropriate command lines or write a shell-program that creates requisite processes with `fork()/exec*()`s. Before matters come to a close, the

following statistics should be compiled:

1. number of readers processed,
2. average duration of readers,
3. number of writes processed,
4. average duration of writers,
5. maximum delay recorded for starting the work of either a reader or a writer.
6. sum number of records either accessed or modified.

When your programs terminate, it is imperative that shared memory segments and semaphores are ***purged***. The **purging of such resources is a *must*** for otherwise, these resources may get ultimately depleted and the kernel may not be able to provide for future needs.

Invocation of your Programs:
Your reader could be invoked as follows:

```
./reader -f filename -l recid[,recid] -d time -s shmid
```

where `./reader` is the name of your (executable) program and its flags may include:

- `-f` designates the binary text-file with name `filename` to work with,
- `-l recid[,recid]` indicates a list of at least one record-ID or multiple record-IDs separated by commas that will be looked up from `filename`; record-IDs have to remain within the range of student records stored in `filename`,
- `-d time` provides the time period that the reader has to "work with the record(s)" it reads in terms of seconds and finally,
- `-s shmid` offers the identifier of the shared memory in which the structure of the records resides in.

Similarly, your writer program could be invoked as follows:

```
./write -f filename -l recid -d time -s shmid
```

where `./writer` is the name of your (executable) program and its flags may include:

- `-f` designates the binary text-file with name `filename` to work with,
- `-l recid` indicates the ID of the record to be updated by the program; this record-ID has to be within the range of records found in `filename`,
- `-d time` provides the time period that the process has to 'stay with the records" it updates in seconds and finally,
- `-s shmid` offers the identifier of the shared memory in which the structure of the records resides in.

You may introduce additional (or eliminate) flags of your choice in the invocation of the above programs. The order with the which the various flags appear is not predetermined. Obviously, you can use any additional flags and/or auxiliary variables you deem required for your programs to properly function.

In general, it would be a good idea to build a logging mechanism possibly in the form of an ***append-only*** **file** that records the work of each player in this group of processes as things develop over time. In this logging mechanism, the activities of each process so far are recorded in a way that is easily understood by anyone who wants to ascertain the *correct concurrent execution* of your programs.