New York University Abu Dhabi
CS-UH 3010
Programming Assignment 3
Due: November 27th, 2021

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. This is essentially the *readers*/*writers* problem discussed in class.

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

1. *studentID*: a 8-character unique identifier,
2. *student name*: up to 20 character string,
3. *grades for courses*: every student can take up to 12 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.
4. *GPA*: the grade point average of the student for the year.

The access protocol for accessing/modifying student records is rather simple: you may have one or more readers at the same time reading records from the binary data file at any time. If a writer shows up with the intention of changing a (random) mark of a student and recalculating the GPA, it has to wait until *all* readers complete their work and exit before it proceeds. Once a writer obtains the right of the way, 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 have access to the *entire* data file of records.

Readers and writers start their work at different and distinct points in time and their operations may 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 the student records" and update their 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. Actually, this is the case with the solution discussed in class. 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`.

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 records so that readers that follow can access the new-introduced content.

Procedural Matters:

◇ Your program is to be written in `C/C++` and *must run* on `NYUAD's Ubuntu` server `bled.abudhabi.nyu.edu`.
◇ You have to first submit your project via `brightspace.nyu.edu` and subsequently, *demonstrate* your work.

◇ Dena Ahmed (`daa4-AT+nyu.edu`) will be responsible for answering questions as well as reviewing and marking the assignment in coordination and collaboration with the instructor.

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/writer may access/update one or multiple records; records accessed/modified are randomly selected. Readers and writers arrive at different times and once they get access to the 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 wants to update records 2 and 6.

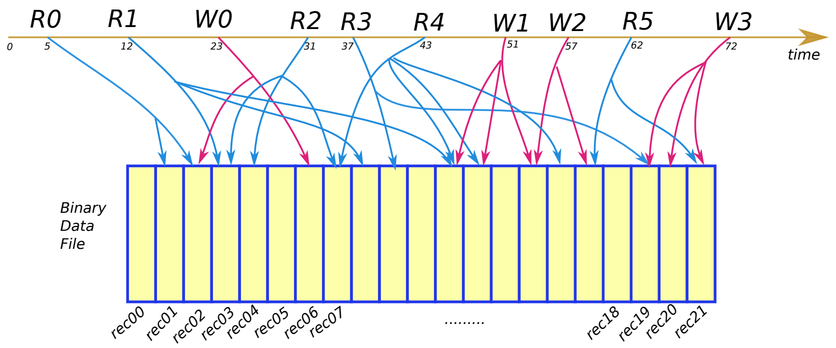

Figure 1: Multiple concurrent Readers/Writers working off a common binary data-file of student records

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 one or more records in the entire file.
- Once a writer starts its work, neither readers nor other writers are permitted to access 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 the records 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 finally exits.
- 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 an array of integers that would provide the process-ids of the readers that are active at any single time instance or the process-ID of the sole writer actively working and modifying records. Objects whose value help you compile statistics at the end of the operation of all your programs, could be held in this region as well. Figure 2(a) depicts the shared memory elements that could help record the state-of-affairs just as the work of *R2, R3* and *R4* is underway. The fixed-size array should feature the process-IDs of processes active in the data file at any specific point in time. Here, the array indicates 3 processes that are active at this time performing look-ups while 1 writer, 2 readers and 7 record-accesses/modifications have already happened. Similary,

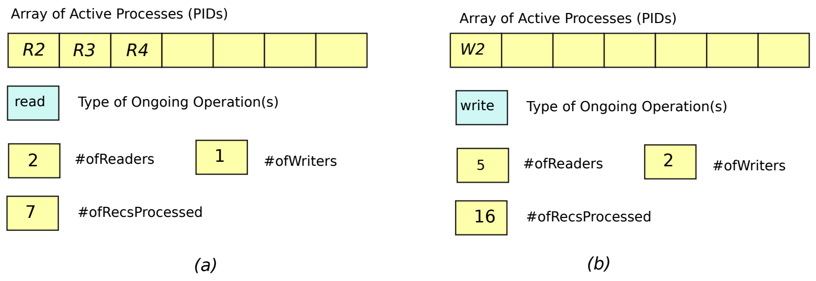
Array of Active Processes (PIDs)
R2
R3
R4
read
Type of Ongoing Operation(s)
2
#ofReaders
1
#ofWriters
7
#ofRecsProcessed
(a)
Array of Active Processes (PIDs)
W2
write
Type of Ongoing Operation(s)
5
#ofReaders
2
#ofWriters
16
#ofRecsProcessed
(b)

Figure 2: An example of shared-memory resident objects that could help in the coordination of *concurrent readers/writers*.

Figure 2(b) shows how the content of structures that help oversee the operation in shared memory, has changed as the work of writer 2 is underway.

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.

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 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 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 -r lb ub -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,
- `-r lb ub` indicates the range (lower-bound and upper-bound) of random records to be looked-up from `filename`; the range always remains smaller or equal to the number 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 -r lb ub -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,
- `-r lb ub` indicates the range of the records to be updated by the program; the range always remains smaller or equal to the number of student records stored 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.

What you Need to Submit:

1. A directory that contains all your work including source, header, Makefile, a readme file, etc.
2. A short write-up about the design choices you have taken in order to design your program(s); a pdf document that is 2-3 pages long would be sufficient.
3. All the above should be submitted in the form of "flat" `tar` or `zip` file bearing your name (for instance `YaserFarhat-Proj3.tar`).
4. Submit the above tar/zip-ball to `brightspace.nyu.edu`

Grading Scheme:

Aspect of Programming Assignment Marked	Percentage of Grade (0–100)
Quality in Code Organization & Modularity	15%
Addressing All Synchronization Requirements	35%
Correct Concurrent Execution for Processes	30%
Use of Makefile & Separate Compilation	10%
Correct Use of In-line Parameters	5%
Well Commented Code	5%

Miscellaneous & Noteworthy Points:

1. You have to use *separate compilation* in the development of your program.
2. If you decide to use `C++` instead of plain `C`, you *should not* use STL/templates.
3. Although it is understood that you may exchange ideas on how to make things work and seek advice from fellow students, sharing of code is *not allowed*.
4. If you use code that is not your own, you will have to provide appropriate citation (i.e., explicitly state where you found the code). Otherwise, plagiarism questions may ensue. Regardless, you have to fully understand what such pieces of code do and how they work.

5. The project is to be done individually as the syllabus indicates and should run on the LINUX server: `bled.abudhabi.nyu.edu`
6. You can access the above server through `ssh` using port `4410`; for example at prompt, you can issue: "`ssh yourNetID@bled.abudhabi.nyu.edu -p 4410`"
7. There is ongoing Unix support through the *Unix Lab* Saadiyat [Uni21].

References

[Uni21] NYUAD UnixLab. `https://unixlabnyuad.github.io/`. With Active Virtual Zoom Session, 2021.