I. Project Objectives:

(1) Understand the concept of UNIX IPC (Inter Process Communication)
(2) Implement and practice inter-process communication and synchronization using message queues
(3) Understand time-sharing UNIX systems
(4) Learn/experience basic system-programming debugging skills

II. Project Narrative:

Since processes do not share their internally produced data with other processes (each process’ memory space is protected from any other processes), sharing data among processes is not trivial. In this project, we implement a message queue (“UNIX message queue”) for two producer processes and two consumer processes. A process, called “master process” is the first process that starts with a program you will develop. After the master process starts, it creates two producer and the two consumer processes as child processes (using “fork” UNIX system call). Each of the two producer processes randomly generates an integer (as unsigned int) at a time and enters a random number to “a message queue”. Each “message queue” is a FIFO data structure managed by the operating system to transfer data from processes to processes (as shown in Figure 1). Each process can enter a piece of data to a message queue. If multiple producer processes enter pieces of data to a message queue, the operating system makes sure that each is entered to a message queue in the first-come-first-serve order. If multiple consumer processes try to remove a piece of data from a message queue, the operating system removes a piece of data from a message queue in the first-come-first-server order while each message queue serves receivers as a FIFO-queue. If a message queue does not have any piece of data, a consumer process that tries to remove a piece of data from a message queue blocks (until a producer enters a piece of data to the message queue). If multiple consumer processes try to remove a piece of data (each), the operating system serves those multiple consumer in the first-come-first-server order.

Figure 1 – two producer and two consumer processes share their data using a message queue
II. Programming Requirements:

(1) Implement a program (“p1_999.c”, where “999” represents the last three digits of your SIUE ID), which performs the following tasks using the program (process) structure (shown in Figure 2):

```
- Create shared memory
- Create message queue
- Create 1st child process
- Create 2nd child process
- Create 3rd child process
- Create 4th child process
- Wait for the master process to create four child processes
- Receive a number from the message queue
- Declare its end
- Terminate itself
- Create shared memory
- Create message queue
- Become the second producer
- Generate a random number
- Send the number to the message queue
- Declare its end
- Terminate itself
- Delete the message queue

*1: declare, create, and attach
*2: define and create
*3: detach and delete it
```

Figure 2 – the required program tasks and structure

(2) For testing purpose, you are expected to use the following modules (their “base implementation” is posted to the CS314 course home):

- Child Process C1 (the first consumer process)
- Child Process C2 (the second consumer process)
- Child Process C3 (the first producer process)
- Child Process C4 (the first second process)
- The function module for “millisleep”
- The function module for “uniform_rand”
- The definition of “message” (for the message queue)

(3) The definition of messages for the message queue is shown below (Figure 3). For our TA to test your submissions within reasonably short time, the data (message) definition is required.
The outputs from your implementation (“p1_999.cpp”) should follow the same outputs (the actual numbers can be different) as the one from the sample outputs (a sample binary executable may be posted to the course home).

Your *.cpp source code file is tested at os.cs.siue.edu using “ce” compiler (will not earn credit, if your submission does not compile or run at os.cs.siue.edu without any change (after it is submitted).

Any required statement/function in the required processes and functions are designated by as label “REQUIRED”, which can not be changed.

The followings are the required activities by the five processes in Project #1 (the master, two producer, and two consumer processes):

1️⃣ The master process creates a shared memory component that consists of:
   (a) “Go_Flag”: as an unsigned integer (0: “not ready to go”, 1: “ready to go”)
   (b) “Done_Flag”: as an array of unsigned integer (“unsigned int”) with four elements (one for each “child process”)
   (c) “Individual_Sum”: as an array of signed integer (“int”) with four elements (one for each “child process”)

2️⃣ The master process initializes the three variables in the shared memory as follows:
   (a) “Go_Flag” = 0 (0: “not ready to go”)
   (a) “Done_Flag”: each element of “Done_Flag” = 0 (0: “not done”)
   (b) “Individual_Sum”: each element of “Individual_Sum” = 0 (initial value)

3️⃣ The master process creates a message queue. The following is the required definition of each message for the message queue (Figure 3):

   ```
   // definition of message
   struct message{
     long mtype;
     unsigned int mnum;
   };
   ```

   Figure 3 – the required definition of each message

4️⃣ The master process creates four “child processes”, one at a time using “fork” system call

   - The first two child processes will be “Consumer1” and “Consumer2”, respectively (in that order).
   - The last two child processes will be “Producer1” and “Producer2”, respectively (in that order).
   - While the master process is creating a total of four child processes (one at a time), each child process waits for “Go_Flag” to be ‘1’ using a “spin-wait”. Each child process loops (using a while-loop) until “Go_Flag” is raised to ‘1’ by the master process.
The master process updates “Go-Flag” from ‘0’ to ‘1’ as soon as it finishes creating a total of four child processes.

The master process waits (by a “spin-wait”) until all four child processes finish (each child process should update its “Done_Flag” from ‘0’ to ‘1’ when a child process finishes.

Each of the two producer processes randomly generates a random number (0 through 100) and enters a randomly generated random number into the message queue. Each of the two producers repeats this for “NUM_REPEATES” times. The label “NUM_REPEATES” should be declared as “pre-compiler directive” at the beginning of your C course code file (i.e., “#define NUM_REPEATES 200”). Each producer should calculate its local check sum as:

\[
\text{Individual\_Sum}[0] = \text{Individual\_Sum}[0] + \text{my\_rand}; \quad \text{(it is assumed that “my\_rand” holds a random number)}
\]

Each of the two consumer processes receives a random number from the message queue and “add” the received random number to its “Individual Sum”.

For example, for the first consumer process (i.e., the third child process), it should be:

\[
\text{Individual\_Sum}[2] = \text{Individual\_Sum}[2] + \text{my\_rand}; \quad \text{(it is assumed that “my\_rand” holds a random number)}
\]

When a child process finishes (either as a producer or as a consumer), it updates its “Done_Flag” from ‘0’ to ‘1’. Then, each child process should just terminate.

For example, for the second producer process (i.e., the second child process), it should be:

\[
\text{Done\_Flag}[1] = 1;
\]

The master process needs to wait until all the four child processes’ “Done_Flag” to be ‘1’ (using a spin wait). When the master process sees that the all four child processes to finish, the master process needs to perform the following activities:

1. Calculates the sum (it is called “final sum”) of “Individual\_Sum” of the two producers and displays the final sum.
2. Deletes the message queue
3. Delete the shared memory

The master process terminates with the following satisfied:

(a) No shared memory or message queue left at os.cs.siue after the master process terminates.
(b) The master process should nicely terminate (i.e., no “crash termination” such as one with “segmentation fault”).
III. Output Requirements:

Each submission should follow the same outputs shown in the figure below (or in the sample output binary executable).

![Figure 4](image-url)

**Figure 4** - the required outputs (no need for you to submit your screenshot, but our TA needs to see this)

IV. Other Requirements:

(a) This project is an individual project. None of the following activities is allowed:
   - Sharing source code files with others as well as showing and e-mailing them to others
   - Sharing program logics and designs (showing, e-mailing, and explaining yours to others)

(b) To avoid “conflicts” of the shared memory and a message queue created by each of you, each of you should use only the assigned shared memory and message-queue keys.

(c) Your *.cpp source code file should be reasonably organized (failing to do the following can result in penalties up to 10% of the project grade in 100-point scale).
   - Program header (your “last 3 digits” (some students need “four digits”), how to compile (if your source code needs more than gcc compiler)
   - In-line comments
   - Proper indentation (to make your source code files readable)

V. Suggestions/Hints:

- **ipcs** – to display the message queues you created (you will see the message queues created by other users in os.cs.siue.edu) for your debugging
- **ipcrm** – to manually delete message queues (if you need to do so when your program fails to delete them)
- **kill** – to kill a process who stops responding
V. Submissions:

(a) Your *.cpp source code files (your name and e-mail address should appear in the program header at the beginning of your source code file) to be e-mailed to the TA (antschn@siue.edu).

(b) The submission (to Moodle) due is at 9:30 A.M. on February 21, 2023.

VI. Late Submissions:

- Penalty of -10% will be given for every 12 hours after the due (i.e., -10% for a submission within the first 12 hours after the due).
- Submission more than 48 hours after the due will not be accepted.
- No “late submission” for early submissions. The extra credit will be given only if all the three programs are completed by the extra-credit early submission deadline. No “partial extra credit” will be given for this extra credit opportunity.

VII. Grading Criteria:

- Compile-time errors: If your submitted *cpp files do not compile using gcc at os.cs.siue.edu due to any compile-time error: 0 ~ 20% of the credit for this project will be given.
- Run-time error: After your source code files are successfully compiled, if your program does not satisfy any requirement: 5% penalty for each minor problem. For any major problem, penalty depends on each such major problem.

VIII. Expectations when a question is asked:

(a) Identify where (in your source code) the problem exists
(b) Describe the symptom(s) of the problem
(c) Describe how the problem happens (always happen, sometime happen, the condition(s) for the problem to happen, etc.)
(d) Describe what you tried (to understand and/or solve the problem)
(e) Stop by Dr. Fujinoki’s office (please no question about programming projects through e-mail).
(f) Bring your source code (either hard copy or soft copy)
(g) No help will be provided on the day before the due (except continuing help that started before the day before the due)

IX. Suggestions/Hints:

(a) UNIX shared memory, fork system call, and message queue:

https://www.tutorialspoint.com/inter_process_communication/inter_process_communication_message_queues.htm
X. Academic Dishonesty

The following conducts (but not limited to them) are considered academic dishonesty, which will result in one of the following sanctions:

(a) For minor cases (e.g., explain a solution/design by non-written format, such as verbally to others): receiving zero credit for this particular project (project #1).

(b) For moderate cases (e.g., showing a solution/design as a memo or sketch for multiple programming issues, or downloading sample programs from the Internet or any other sources after they are modified): receiving zero credit for the entire project grade (not only for project #1).

(c) For major cases (e.g., e-mail source code file or copy/exchange source code files with others, or downloading a whole program source code file from the Internet or any other sources and resubmit it with minor modifications): receiving an F grade for this course.

The bottom line: you are expected to demonstrate your learning regarding how to handle similar programming projects using only technical references that explain the syntax of the software tools or/and only basic samples that describe the syntax. If you are not sure, contact Dr. Fujinoki, instead of making your own assumptions.