CECS 326: Group project[1]

Objectives: Interprocess communication and basic coordination using message queue[2]

This assignment is to be completed in a group[3] of 3 or 4 students. I recommend that each group selects a leader. The requirements of the system is very complex because there are several software dependencies, a great number of possible “paths of execution” and intermittent software bugs. Get started as soon as message queue is covered in lab; your group will need the entire time: testing, redesigning & recoding. If you need assistance with the assignment, you should ask for help. If you should need assistance with a difficult member of your group, you can & should speak with me in private about issues confronting your group immediately.

Software requirements:

Design and implement an “event” monitoring & reporting system. An event is a random number that is divisible by some pre-assigned "marker" value and the random number is generated internally by the software component. When an event is observed by a component, the phenomenon is reported to other components.

This assignment consists of 5 separate components/programs that execute concurrently on the same machine. Three of the programs are sending data to 2 receiving programs. When a *sender* observes its event, it sends its *receiver* the sender identity & the observed value (among other necessary items). Each *sender* generates random numbers (32-bit value). Each *sender* is pre-assigned with its own marker integer: 251, 257 or 997 and behaves slightly differently. First, the 997 *sender* sends each event to all *receivers* and requires acknowledgement for each message from both receivers before it continues execution. Also, this *sender* terminates when it gets/observes a random number smaller than 100.

The other 2 *senders* do not accept any ack message. The 251 *sender* only reports its event to one *receiver* & terminates on a 'kill' command[4] from a separate terminal. Like the second *sender*, the last 257 *sender* only notifies one *receiver* & terminates when its *receiver* stops receiving event notification (see below).

Each *receiver* repeatedly gets a message & displays the value and *sender*'s identity. The first *receiver* accepts messages from 251 & 997 *sender*s. This *receiver* only terminates after both of its *sender*s had terminated. The second *receiver* gets messages from 257 &

---

[1] This project can be completed after the lab lecture on the sample program on message queue.
[2] This is an abstraction and simplification of a monitoring system. For instance, one may require a system that monitors wildlife or marine life. One sensor counts elephants in the wild, a second detects zebras, and a third monitors tigers.
[3] If you need assistance with finding a group, let me know.
[4] Your professor will supply a patch code to be integrated with this 251 sender. The kill signal (10) that is entered will trigger the patch code to execute, which sends one (last) message to its intended recipient before before the sender terminates. See my web page for addition information.

997 *sender*s. The second *receiver* terminates after it has received a total of 5000 messages. (Also, remember to send the ack to 997 *sender*.)
There are a number of ways to design[5] (e.g., handling non-existing queue exception, protocol for exchanging messages) & implement this system. Each design likely has advantages & disadvantages. Therefore, it is important that students explain the rationale of their chosen design & implementation.

Programming notes:

1. Students will construct 5 separate programs (fewer is possible too). The marker values can either be hard coded or user input or command line switch. Each program needs to execute with the maximum degree of concurrency[6], without unnecessary delay.
2. Student needs to design the content of the message structure and the communication protocol. This can be tricky. That is, this document only specifies data items that are relevant to the tasks the programs perform. Student may include other pieces of information and/or other messages that are needed to ensure that the system operates correctly and cohesively. Nonetheless, minimize overhead. Remember to include the `mtype` variable as the very first field of the message structure and the message structure must be a fix-sized object; it cannot include pointers, linked lists, string object, vectors, etc.
3. Assume an infinite message storage capacity & lossless communication (obviously not true in real life).
4. State and document additional constraints and assumptions in your design and implementation.
5. The programs should not execute in any predictable order, unless absolutely necessary, e.g., the program which creates the message queue must start first or the program that deallocates the message queue must be last to exit (although it may finish its tasks before the others).
6. All interprocess communication will be accomplished with a single message queue. Remove the queue automatically when the last program exits. The queue should be empty when it's deleted.
7. Avoid using the *sleep( )* function or do not use a busy wait loop[7] – a loop structure whose primary purpose is to halt or sequence program execution until a certain condition is satisfied.
8. Race conditions (e.g., using the queue before it is created, deleting the queue too soon, taking a message that is intended for another recipient, etc.) are to be avoided.
9. It is easiest to execute each program in its own separate terminal/window.

---

[5] The above specification is purposely general, in order to allow students to explore a broad range of options in choosing a design and implement for this system. Therefore, a student may produce a slightly different programming solution. Consequently, some implementations will be more responsive, more reliable, more scalable, more robust, and easier to maintain than others. It would be appropriate to document, explain, and justify your design choices and assumptions. When in doubt about the requirements of your programs, please ask questions.

[6] Allowing the highest number of processes to execute concurrently.

[7] Typically present when the programmer uses the prohibited IPC_NOWAIT option.

Submit hard copy of your source code & documentation, a cover page with description of the software, and your information (names of each member of the group, date, the course number, email address). Show your instructor the execution of your programs.

Note on working with message queue:
Students can learn more about a system call using Linux command 'man 2 syscall' where syscall is one of: msgget, msgsnd, msgrcv, or msgctl. The msgget( ) function as shown in the sample program B would return an error (-1) if the queue already exists -- not deleted on previous execution. This situation may be handled with two Linux terminal commands: 'ipcs' and 'ipcrm msg msqid' -- see 'man ipcs' for more info. To remove the unallocated queue, use the ipcrm command with the msqid value from the ipcs command.

Below is a sample output of the ipcs command. When a garbage queue is present, there would be an entry under the Message Queues section.

```
$ ipcs
------ Shared Memory Segments --------
key      shmid  owner  perms  bytes    nattch  status
------ Semaphore Arrays --------
key      semid  owner  perms  nsems
------ Message Queues --------
key      msqid  owner  perms  used-bytes       messages

$
```