CECS 326-01 Assignment 2 (10 points)

Due: 10/15/2024 online on Canvas

When a user launches a program for execution, the OS creates a user process to execute it. Many OS, including Linux, provides the mechanism for a process to create child processes to run concurrently with the parent process which creates them. Linux provides such support with system calls **fork**, **exec and wait**. These concurrent processes may need to communicate among them. One way Linux supports interprocess communication is message queue. Using the System V implementation, a message queue must first be acquired from the operating system by calling **msgget**. Control operations, e.g., remove, can be performed on an existing message queue by calling **msgctl**. Processes with appropriate permissions may send and/or receive messages via the message queue by calling **msgsnd** and **msgrcv**. A process may obtain its own PID by calling **getpid**. Please consult the man pages of these system calls for details.

In this assignment you will make use of what you learned about the Linux message queue mechanism in class, and make use of the fork(), exec() and wait() system calls to create child processes and control the child process' execution. For this assignment you need to write three C++ programs named *master.cpp*, *sender.cpp*, and *receiver.cpp*, which should be compiled into executables *master*, *sender*, and *receiver*, respectively. Be sure that *sender.cpp* and *receiver.cpp* are themselves C++ programs with main functions, NOT classes to be included in *master.cpp*. Together they should do the following:

When *master* executes, it should first output a message to identify itself. It should then acquire a message queue from the kernel, followed by creating one child process to execute *sender,* and a number of child processes to execute *receiver*, with the number of *receiver*s passed to it from the commandline argument. The *sender* process should obtain the message queue id and the number of receivers in the exec() system call that launches its execution. Each *receiver* process should receive the message queue id and its receiver number (1, 2, …, etc. based on the order of the process' creation) in the exec() system call that launches its execution. The *master* process should output its process ID, the message queue ID, and information about each child process that it has created, including if it is to serve as a sender or receiver, receiver number, and their respective PID. The *master* then waits for all child processes to terminate, then removes the message queue and exits.

master should run with the command below and produce the following sequence of output:

./master x [*x* is the number of receivers]

```
Master, PID xxxxx, begins execution
Master acquired a message queue, id zzzzz
Master created a child process to serve as sender, sender's PID yyyyy
Master created a child processes to serve as receiver 1
Master created a child processes to serve as receiver 2
Master created a child processes to serve as receiver 3      [the number of receivers is x from the aommandline
argument]
Master waits for all child processes to terminate
Master received termination signals from all child processes, removed message queue, and terminates
```

When *sender* executes, it should first output a message to identify itself, and show the message queue id it obtained from the exec() system call that invokes its execution. It should then perform the following sequence of action for a number of times (you may choose the number of times in your run):

1. prompt user to enter a line of input and indicate which receiver the input line is for
2. send the input line to the message queue and tag the message with the intended receiver number

sender then goes into a loop to wait for acknowledgements for all sent messages. All acknowledgement messages should be tagged with a unique number, say, 999 [Make sure the tag number is not zero. A zero for msgtype means no tag.] The acknowledgement message should be in the form of: Receiver y acknowledges receipt of message. [y is the receiver number.]

sender should produce the following sequence of output when it executes:

```
Sender, PID xxxxx, begins execution
Sender, PID xxxxx, received message queue id zzzzz through commandline argument
```

sender then produces the following sequence of output for each message to be transmitted to a receiver:

```
Sender, PID xxxxx: Please input your message
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx              [this is user input message]
Sender, PID xxxxx, which receiver is this message for?
y                                                   [y is user input for the intended receiver]
Sender, PID xxxxx, sent the following message into message queue for receiver y:
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

sender then produces the following line of output for every acknowledgement from a receiver:

```
Sender, PID xxxxx, receives message: Receiver y acknowledges receipt of message
```

When *receiver* executes, it should first output a message to identify itself, and show the message queue id and receiver number it obtained from the exec() system call that invokes its execution. It then retrieves a message sent by a sender from the message queue and outputs the message, then sends an acknowledgement message into the message queue before terminating.

Each *receiver* should produce the following sequence of output when it executes:

```
Receiver y, PID yyyyy, begins execution             [y is the receiver number]
Receiver y, PID yyyyy, received message queue id zzzzz through commandline argument
Receiver y, PID yyyyy, retrieved the following message from message queue
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx              [this is message tagged with y retrieved from message queue]
Receiver y, PID yyyyy, sent acknowledgement message into message queue        [Acknowledgement message should be in the form: Receiver y acknowledges receipt of message.  Note that y is the receiver's receiver number.]
Receiver y, PID yyyyy, terminates
```

Notes:

1. The display of the above sets of output may interleave.
2. You may want to experiment with 1 sender and 1 receiver initially.
3. The use of message tags enable the distinction between a message sent into the message queue by a sender for a particular receiver and an acknowledgement message sent by a receiver.
4. The program must run successfully on Linux.

Submit on Convas the following:

1. The source programs *master.cpp*, *sender.cpp*, and *receiver.cpp*;
2. A screenshot that shows successful compilation of *sender.cpp* to *sender*, *receiver.cpp* to *receiver*; and *master.cpp* to *master*;
3. A screenshot that shows successful run of *master*, which will create child processes to execute *sender* and *receiver*; producing outputs as described above;
4. A cover page that provides your name, your student ID, course # and section, assignment #, due date, submission date, and a clear program description detailing what the programs are about. Format of the cover page should follow the cover page template posted on Canvas.
5. The programs must be properly formatted and adequately commented to enhance readability and understanding. Detailed documentation on <u>all</u> system calls are especially needed.