Birzeit University - Faculty of Engineering and Technology
Electrical & Computer Engineering Department - ENCS531
Real-Time Applications & Embedded Systems - 2^{nd} semester - 2019/20

Project #1
Semaphores, Message Queues & Shared memory under Unix/Linux
Due: April 28, 2020

Instructors: Dr. Ahmad Afaneh, Dr. Hanna Bullata

Coronavirus simulation

We would like to create a multi-processing application that simulates a country ravaged by the coronavirus pandemic. The objective of the simulation is to see under what conditions the coronavirus patients have a better survival rate. As you might expect, we'll make the simulation simple by considering only a limited number of symptoms (parameters). We'll consider the following parameters:

- **Number of available doctors**: Must be a positive number bigger than 0 and less than 10,000.
- **Coronavirus illness severity of patient**: We'll classify the patients according to the 3 main symptoms:
  1. Fever,
  2. Cough,
  3. Shortness of breath.

  Patients can have any or all of the above 3 symptoms and these are attributed to patients at random. Thus, worst severity is 3 (all 3 symptoms are present), best severity is 0 (no symptoms).
- **How long have they been ill**: That can be a random number picked from an input interval (e.g. [1 day .. 10 days]). The more the patient has been ill, the bigger is the illness severity.
- **Patient medical history**: We'll consider the following factors:
  1. Blood hypertension,
  2. Heart/respiratory system diseases,
  3. Cancer.

  Again, patients can have any or all of the above 3 diseases that are attributed to patients at random. Assume if the patient has blood hypertension, add 1 point to the severity, if he/she has heart/respiratory system diseases, add 2 points and if the patient has cancer, add 3 points.
- **Age of patients**: We consider that the older is the patient, the higher is the severity.
- **How long has the patient been waiting**: Patients might not get immediate attention from doctors if none of them is available. The more the patient waits, the more his/her severity increases.

Since the above are all input arguments to the simulation, it'll be a good idea to put them in a text file and pass the text file to the application once run. An example of input parameter file can look as follows:

```
NUMBER_OF_DOCTORS 100
FORK_NEW_PATIENT  [1 .. 10]     # in seconds
COVID_19_FEVER    [0 .. 8]      # If random value more than 4, TRUE, else FALSE
COVID_19_COUGH    [0 .. 8]      # If random value more than 4, TRUE, else FALSE
COVID_19_BREATH   [0 .. 8]      # If random value more than 4, TRUE, else FALSE
BLOOD_HYPER       [0 .. 8]      # If random value more than 4, TRUE, else FALSE
HEART_RESP        [0 .. 8]      # If random value more than 4, TRUE, else FALSE
PATIENT_AGE       [10 .. 110]
.
.
.
```

You can add to the above file as you see fit.

The sequence of operations should be done as follows:

1. A parent process (*parent*) should be responsible for forking a certain number of doctors. The number of doctors should be picked from the above input file.

2. Upon being forked, each *doctor* will create its own message queue. The number of message queues should be equal to that of the doctors. The doctors processes should register their message queue IDs in the shared memory.

3. The parent process will fork patients every random number of seconds. That number of seconds should belong to the interval stated in the input parameter file.

4. Once a *patient* process is created, it has to decide on the patient age and severity level. Afterwards, the patient should register its PID in a shared memory that the doctors processes can access.

5. Each *doctor* process should read a pid for one of the waiting patients from the shared memory. The access to the shared memory should be exclusive so it has to be protected by a semaphore. If no patient is waiting yet, the doctor should sleep for a period of time before trying again. That period of time can be user-defined and thus added to the input parameter file. Otherwise, the doctor process should do the following:

   – The doctor process should write its message queue ID in a location accessible by the patient process.
   – The doctor process sends a signal to the patient and waits for a message to be sent by the patient on its message queue for a certain amount of time.
   – If a message is not received by the doctor after a number of seconds, the doctor process jumps to the next patient in the queue.
   – If a message is received by the doctor process, it sleeps for a period of time. Afterwards, it sends the message "show-up" to the patient.

6. When the patient process receives a signal from the doctor process, it reads the message queue ID from the shared memory and sends the message "Iam sick" to the doctor process.and send all the patient attributes to the doctor process and waits for a reply. If a reply message is not received by the patient after a delay, the patient process dies. In addition, if the doctor process decides the patient is in a desperate situation, the patient process dies. Else, the patient recovers and exits peacefully.

7. While a patient process is alive and waiting for a doctor, it has to increment its severity. The more the patient process waits, the higher is the severity level. If the severity exceeds a threshold, the patient process dies.

8. The simulation application ends when the number of deceased patients gets higher than a user-defined threshold.

Procedure

- The parent's job is to prepare the grounds for its children, mainly create the semaphore sets, shared memory before forking the doctors and patients processes
- the parent process must monitor how the patient processes exit the application. If they die peacefully, it means the patients have recevered. Otherwise, that means the patients dies.
- The parent should clean the environment once the simulation is over (e.g. remove the semaphores sets, shared memory, message queues, etc).

What you should do

- Write the code for the parent process in file `parent.c`, for the doctor processes in file `doctor.c` and for the patient processes in file `patient.c`.
- Compile and test your program.
- Check that your program is bug-free. Use the `gdb` debugger in case you are having problems during writing the code (and most probably you will :-). In such a case, compile your code using the `-g` option of the `gcc`.
- Send the zipped folder that contains your source code and your executable before the deadline. If the deadline is reached and you are still having problems with your code, just send it as is!