COMPUTER NETWORK (CS F303)
LAB-SHEET – 5
Topic: Socket Programming- Concurrent Server

Learning Objectives:

Creating concurrent TCP server (handling multiple clients) using fork() system call

Handling multiple clients at the same time

There are two main classes of servers, iterative and concurrent. An iterative server iterates through each client, handling it one at a time. A concurrent server handles multiple clients at the same time. The simplest technique for a concurrent server is to call the **fork** function, creating one child process for each client. An alternative technique is to use **threads** instead (i.e., light-weight processes). In this lab, we consider fork based technique only.

A typical concurrent ECHO server has the following structure.

Variable declaration, socket creation, and

```
while(1){
      newSocket = accept(sockfd, (struct sockaddr*)&newAddr, &addr_size);
      if(newSocket < 0){
      exit(1);
                     }
      printf("Connection accepted from %s:%d\n",inet_ntoa(newAddr.sin_addr),
ntohs(newAddr.sin_port));

      if((childpid = fork()) == 0){
      close(sockfd);

      while(1){
            recv(newSocket, buffer, 1024, 0);
            if(strcmp(buffer, ":exit") == 0){
            printf("Disconnected from %s:%d\n",inet_ntoa(newAddr.sin_addr),
ntohs(newAddr.sin_port));
      break;}
      else {
            printf("Client: %s\n", buffer);
            send(newSocket, buffer, strlen(buffer), 0);
            bzero(buffer, sizeof(buffer));
           }
      }
  }
 }
```

```
        close(newSocket);
```

When a connection is established, accept returns, the **server calls fork**, and the child process services the client (on the connected newsocket. The parent process waits for another connection (on the listening socket sockfd. The parent closes the connected socket since the child handles the new client.

Exercise #1

Using this code, you can extend your TCP Echo server created in Lab3 to handle multiple clients simultaneously.

Exercise#2

Develop a simple **key-value () store** using TCP sockets where clients connect simultaneously to the server for inserting and retrieving the key. Your application must support the following features:

- The client takes user input to **get/put/delete keys** and passes these requests over the server.
- The server maintains a data structure of **key-value** pairs in a file (**database.txt**) as the persistent database and returns a suitable reply to the server.
- When the request from the client is of the form **"put key value",** the server must store the key-value pair in its data structures, provided the key does not already exist. Put requests on an existing key must return an error message.
- When the client sends a request of the form **"get key"**, the value stored against the key must be returned to the client. If the key is not found, a suitable error message must be returned by the server.
- The client should be able to issue a delete request (on an existing key) as follows: **"del key"**.
- For all requests that succeed, the server must either return the value (in case of get), or the string **"OK"** (in case of put/del).
- You may assume that all keys are integers and the corresponding values are strings. When the user inputs "Bye", the server should reply **"Goodbye"** and close the client connection.
- Upon receiving this message from the server, your client program terminates. However, the server must keep the database in a file for subsequent run of client program.
- Note that your server must be able to communicate with multiple active clients at a time. When multiple clients talk to the server, one client should be able to see the data stored by the other clients.

xx—oo—xx