Operating Systems — *CS 450 — Fall 2013*

Project 1 – Phase 1 — Due on Monday, October 28 at 12:00 noon

For phase one of this project, you are to implement the following features.

- Implement a server that responds to one client only. The server, after it accepts a socket connection from the client, creates a suitable thread to handle that connection. For example, the server, after having accepted a connection from a client and having stored the socket descriptor of that connection in clientSD, could do:

```
// The following four variables are global to all server functions.
char clientMessage[ ... ];
bool hasInfoToWrite = false;
pthread_mutex_t readerMutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t readerCond = PTHREAD_COND_INITIALIZER;

...

char command[5];

read( clientSD, command, 4 );

command[4] = '\0';  // assume we did successfully read 4 bytes.
pthread_t tid;
int *pToClientSD = new int;
*pToClientSD = clientSD;
if( strcmp( command, "read" ) == 0 )
      pthread_create( &tid, NULL, readerProcessThread, pToClientSD );
else if( strcmp( command, "writ" ) == 0 )
      pthread_create( &tid, NULL, writerProcessThread, pToClientSD );
else if( strcmp( command, "join" ) == 0 )
      pthread_create( &tid, NULL, commandProcessThread, pToClientSD );
else
      // send 0 to the client...
```

  The code for the three functions in the server module could be:

```
void *commandProcessThread( void *cFD ) {
  int clientSD = *(reinterpret_cast<int *>(cFD)); // Recast to the original datatype

  // read the client's proposed clientID here and store it in a
  // global variable so that the readerProcessThread and
  // writerProcessThread can have access to it.

  while( true ) {
    char buf[ 5 ];

    // read 4 bytes and take appropriate action.
    read( clientSD, buf, 4 )
    // assume that read was successful
    buf[4] = '\0';
    if( strcmp( buf, "lscr" ) == 0 ) {
      // send a list of available chatrooms to the client.
    } else if( ...)
    ...
  }
}
```

```
void *readerProcessThread( void *cFD ) {
  int clientSD = *(reinterpret_cast<int *>(cFD)); // Recast to the original datatype

  // read the client-id of the client process using the socket
  // descriptor, clientSD. For this phase, since we deal with one
  // client only, we will not use the client id. Just make sure you
  // are able to read it.

  while( true ) {
      pthread_mutex_lock( &readerMutex );
      if( hasInfoToWrite ) {
            // write the contents of clientMessage to the socket whose
            // descriptor is stored in clientSD and set hasInfoToWrite
            // to false.
      } else {
          pthread_cond_wait( &readerCond, &readerMutex );
      }
      ...
      pthread_mutex_unlock( &readerMutex );
  }

}

void *writerProcessThread( void *cFD ) {
  int clientSD = *(reinterpret_cast<int *>(cFD)); // Recast to the original datatype

  // read the client-id of the client process using the socket
  // descriptor, clientSD. For this phase, since we deal with one
  // client only, we will not use the client id. Just make sure you
  // are able to read it.

  while( true ) {
      short msgLen;
      // Read the number of bytes in the client-id message.
      read( clientSD, (char *) &msgLen, sizeof( short ) );
      msgLen = ntohs( msgLen );       // convert to local-host byte-order
      read( clientSD, clientMessage, msgLen ); // of course msgLen has to be less
                                               // than the number of elements in
                                               // clientMessage.
      clientMessage[ msgLen ] = '\0';
      pthread_mutex_lock( &readerMutex );
      hasInfoToWrite = true;
      pthread_cond_signal( &readerCond );
      pthread_mutex_unlock( &readerMutex );
      ...
  }
  ...
}
```

- Write the implementation for the three client processes. These are three independent programs. The *command process* could be invoked like this.

  ```
  ./commandProcess.x ip-of-server port-number-of-server
  ```

  The *command process*, after having established a socket connection with the server using ip-of-server and port-number-of-server, and having stored the socket-descriptor in serverSD, repeatedly reads user's commands from the standard-input and takes appropriate action. For example, after having read *join* from the standard-input, it could do:

```
write( serverSD, "join", 4 );
char buf[1];
read( serverSD, buf, 1 );
if( buf[0] == '0' )
   // chat-server, for some reason, rejected this client from joining.
else if( buf[0] == '1' ) {
   // chat-server accepted the join command. Propose a client-id to
   // the server.
   const char *clientID = "TheChatMonger";
   short idLen = strlen( clientID );
   idLen = htons( idLen );
   // send the length of the message first.
   write( serverSD, (char *) &idLen, sizeof( short ) );
   // and now the message.
   write( serverSD, clientID, strlen( clientID ) );
   ...
}
...
```

Your *command process* should read from the standard-input, and identify every command that it is programmed to process (in the above, I showed how to react to the *join* command.) Aside from the *join* command, once the *command process* reads a command, from example *subs*, it should send it to the server and print the status that the server returns to the standard output. The server thread that communicates with the *command process*, for this phase and for commands other than *join*, would just send a 0 or a 1 back to the client.

The task of the *reader process* and the *writer process* are simpler. Initially, each of these two processes establishes a socket connection with the server, identifies itself by sending it's identifying command, and sends the client-id that the *command process* established with the server. After this step, the *writer process* reads from the standard input and writes what it reads directly to the socket and the *reader process* reads from the server socket and echoes what it reads to the standard output.

Important Notes

1. In the above code-segments, I have used system-calls *read* and *write* for the demonstration of the ideas only. As we have discussed in the class, when you attempt to read n bytes from a socket, read may return m bytes, where $0 < m \leq n$. Therefore, you may have to read multiple times, store the bytes as you read them in a buffer, until you read the entire n bytes (note that the read system-call could return zero at any time during this process.) The same principle applies to the *write* system call.

2. I have hard-coded the client-id for simplicity. The *command process* should read the client-id from the standard input, send it to the server, and if the server rejected it, inform the user and read another client-id to send to the server. this process should continue until the server accepts the client's proposed client-id.

3. The naming of the readerProcessThread and the writerProcessThread in the implementation of the server are based on the client's view point. That is, the *reader process* reads from the socket that it establishes with the readerProcessThread while readerProcessThread actually write to this socket. The naming could become a bit confusing.

4. The readerProcessThread and the writerProcessThread should use a *bounded-buffer* to communicate with one another. However, for this phase of the project, I have used a single

buffer and have assumed that the your *writer process* sends a message to the server and the server bounces it to the reader process, which displays it to the standard output. For this to work, I have used a mutex and a condition-variable. Feel free to experiment with alternatives for mutual exclusion: we studied a lot of them. ;-)