Program 6: Musical Chairs :: Sockets

CSCI 4547 / 6647 Systems Programming
Fall 2020

1 Goals

1. To practice coordinating multiple processes by using a protocol.
2. To use Internet sockets and TCP/IP.
3. To implement a communication protocol that handles multiple clients competing for a resource.
4. To create a makefile that compiles and links both client and server.

2 Instructions

Implement the same musical-chairs logic as for program 5, but do it with sockets. There are three big changes from the thread model: communication is more difficult, synchronization is easier, and the mom code and the Kid code will be compiled separately, using a common header file.

- Mom (the server) will receive N, the number of kids playing the game on the command line. She will set up a welcome socket at port 1234 for the initial communication with the kids (remote clients). Mom will listen for contacts on the welcome socket.
- Each child must be started up separately. It will create a socket through which messages will flow to and from from mom, then try to contact mom on port 1234.
- Mom will pass the incoming kid's connection from the welcome socket to a worker socket. When N kids have arrived, the welcome socket can be closed and the game can start. Since the welcome socket must not be part of the polling loop, setting up the polling structure is a bit less complicated than in my example program.

Servicing the sockets. This program's use of sockets falls into two phases. In phase 1, mom starts up and kids arrive at the party. Nothing else happens until all the expected kids are there. If a kid-connection dies during the game, it is OK for mom to end the game by calling fatalp().

In phase 2, mom and the kids play multiple rounds of the game, with one kid being eliminated at each round. A kid who is eliminated should close its socket, and mom should change the polling structure for that socket to tell the polling loop to ignore it. This is done by changing the fd for that socket to the negative of what it was originally:

```
socks[k].fd *= -1;
```

At the end, mom and one kid remain, and they terminate after mom sends a prize to the kid. She should then close the worker sockets:

```
for (int k=0; k<nKids; ++k) {
    if (socks[k].fd < 0) continue;
    close( socks[k].fd ) ;
}
```

Note that this loop ignores sockets that are already closed.

2.1 The shared header file.

Both the mom and kid modules must include a header file that gives the port number where mom will listen for contacts. This file may also contain a description of the protocol messages, in the form of comments and an enum type, depending on how you choose to implement the game. Note: Objects are normally NOT defined in header files. If a header file that contains an object is included in more than one compile module for THE SAME application, the application will not link.

Four of the messages in the protocol have a command followed by data. The other message have just the command.

- HELLO: From kid to mom as soon as the connection is made. A HELLO message is a string of the form `"HELLO bobby"`, where bobby is the kid's name.
- GETUP: From mom to kid at beginning of each round. A GETUP message is a string of the form `"GETUP 8"`, where 8 is the current number of chairs.
- SIT: From mom to kid when she turns off the music. Example: SIT
- WANT: From kid to mom when kid gets the SIT message. A WANT message is a string of the form `"WANT 3"`, where 3 is the chosen chair number.
- ACK: Your choice is approved – sit in it. Example: ACK
- NACK: Bad choice – somebody else already took that chair. A NACK message is a string such as `"NACK 01001101"`, which says that chairs 1, 4, 5, and 7 are still available.
- QUIT: You lost; time to quit. (From mom to the chairless kid when there are no chairs left.) Example: QUIT
- PRIZE: mom sends the surviving kid a message about winning. Example: PRIZE

2.2 Stream input and output.

The SOCK_STREAM type of socket provides a sequenced, reliable, two-way connection through which a sequence of ASCII bytes can be sent and received. We are accustomed to using a high-level stream interface that takes care of skipping whitespace, the tedious and picky ASCII-to-integer conversion (after a read) and formatting (before a write). Sockets work with low-level I/O, which transmits the bytes unchanged and does not do conversions, whitespace management, buffering, etc.

It is *possible* to do all message processing with low-level I/O. However, it will save a great deal of programming and debugging time to open streams on the sockets and use normal high-level I/O. The `fdopen()` function opens a C-stream on top of the socket. For 2-way communication, you need two streams on each socket:

```
kidIn = fdopen( socks[k].fd, "r" );
kidOut = fdopen( socks[k].fd, "w" );
```

where kidIn and kidOut are class members of type FILE*. The code in the kid is somewhat easier since the kid has only one mom, not an array of moms. Having set this up, you can use the normal C I/O calls to `fprintf() and fscanf()`.

Note well:

- Write a newline character at the end of each protocol message. This will help to parse the messages.

- After using `fprintf()` to write to a socket, use `fflush()` to send the message immediately through the socket; otherwise, it will stay in the buffer on the sending end until the buffer is full.

3 Code Files and Classes

I wrote and debugged this program twice: once with all the code in two files (mom.cpp and kid.cpp). Then I wrote it again using three files per module. There is no question that the second result is better, easier to understand, and easier to debug. So my suggestion is that you start from the beginning with these files:

- socktypes.h: typedefs for system types and, as comments, definitions of the socket data structures.
- protocol.hpp: the port number and, if needed, the list of messages used in the protocol.
- mom.cpp: the main program for the mom module. Makes all the connections then instantiates the MomLogic class and calls its run function. Principle: separate the connection logic from the business logic.
- momLogic.hpp: Defines a class that contains chairs, counters, an array of Players, C-streams, buffers, and everything else needed to communicate between one of the MomLogic functions and another.
- momLogic.cpp: Implements the functions, instantiates KidLogic, and calls its run function.
- kidLogic.hpp: Header file for the KidLogic class. Defines the name and state of the kid and two C-streams.
- kidLogic.cpp: Implements a function for each part of the protocol plus a function that writes messages, one that reads and parses messages, and a function that looks at the command in a message and calls one of the protocol functions.

4 Mom's Life.

At the beginning of the game, mom reads N from the command line and uses it to create an array of kid-sockets. The array and N will later be passed to the MomLogic constructor and stored in the MomLogic class. The command line for starting up mom looks like this:

```
~> mom 20
```

where 20 is the number of kids that mom will accept. Limit N to 25 or fewer.

When all the kids have arrived, close the welcome socket: `close( welcomeSock );` , instantiate MomLogic, and call its `run()` function. When MomLogic.run() returns, close all the sockets.

The TCP connection. One of the difficult parts of debugging a socket program is that the TCP connection stack on a Linux machine can take up to 2 minutes to clear the port number out of its tables after an abnormal termination of the server. Abnormal endings of all sorts happen constantly during testing, and until the TCP connection is cleared, you can't rerun the code. The solution is to initialize mom's welcome socket so that the port number is reusable. Here is the code; you need that trueVal variable because the argument to setsockopt must be passed by address:

```
int trueVal = 1;
status = setsockopt(welcomeSock, SOL_SOCKET, SO_REUSEADDR, &trueVal, sizeof(int));
```

```
if (status < 0) fatalp("Can't set reuse socket option");
```

When you call `listen()`, supply a queue length that is bigger than 1:

```
    listen(welcomeSock, 10).
```

Otherwise, if multiple children arrive at once, only the first will be accepted, the rest will terminate without connection.

4.1 MomLogic's Preparation and Cleanup.

Mom's main has an array of polling structures, one for each kid. It must be a parameter to the MomLogic constructor. The MomLogic constructor must allocate an array of N chars to represent the chairs. A '0' will represent an occupied chair and a '1' chair is available. The last slot of the array is a null terminator. In addition, MomLogic also needs to create an array of Players. This type can be declared in the private area of MomLogic. An array of N Players must be a class member for MomLogic.

```
struct Player {
    string name = "??";
    bool alive = false;
    FILE* kidIn;   // Stream for kid to send to mom.
    FILE* kidOut;  // Stream for mom to send to kid.
};
```

The boolean variable is used in the polling loop to skip connections that have been broken. With this information, it is possible to write the logic without swapping the positions of anything in the polling structure. This, together with the kid's name, makes it much easier to debug the game. The two streams are needed to enable high-level I/O.

Use the MomLogic destructor to free any dynamic memory that you create with new anywhere in the class.

4.2 MomLogic's run() Function.

This function is called from mom's main. It finishes the setup phase of the game. It must read messages from all of the kids and store their names in the Player array for future reference. Then mom can enter a loop to setup and play rounds of the game until there is only one remaining kid. Finally, she sends the prize to the last kid.

4.3 One Round of the Game: Details

To play a round of the game, initialize the round, then stop the music.

- `initRound()`
  - At the beginning of each round, mom will shorten the chair array so that there is one fewer chair than kid. She initializes all those chairs to '1' and stores a null character in the chair at subscript $nAlive - 1$.
  - Then she sends the GETUP message (with the current number of chairs) to all the kids.
- `stopTheMusic()`: After sleeping a second or two, mom will send the SIT message to all the kids. Then she enters a polling loop.

  - At this stage, each kid will send back a WANT message, followed by a seat number. Mom needs to poll the set of kid sockets to read and respond to their requests. Each round of polling requires you to check each of the sockets, skipping those that belong to dead children.
  - The easiest way to do this is to have the loop that checks the sockets in a separate function. This avoids the complexities of managing a nested loop.

- `checkSockets()`: Called from stopTheMusic(). Mom needs to check each socket once and handle requests. If there are no errors, there will be at least one waiting request, but there may be more. However, some kids will have requests on this round and others will not.
  - In this loop, it is important to skip any kid-sockets that belonged to dead kids. That is why the Player struct has a field called "alive". If a kid is not alive, don't try to read from its socket.
  - If kid k's socket has no requests, go back to the top of the loop and check the next kid.
  - Otherwise, get the message on socket k and handle it. You need to keep track of how many chairs are available at any given time (or how many kids have found seats).
  - Mom will reply with an ACK if that chair is available, then she stores a 0 in that location of the string. She should also decrement a chair-counter so she knows when all the chairs are full.
  - She will reply with a NACK if the chair is already occupied, followed by a string representing the chairs that are still available.
  - The round ends when mom receives a chair request from kid k, but sees that all chairs are filled. She must tell kid k to QUIT and store `false` in the "alive" field in the corresponding kid struct.

5 A Kid's Life.

Argc should equal 3 for the kid-main function: ~> `kid hostname carrie`
The kid must make a socket connection to the host, then instantiate KidLogic and call its run function. The KidLogic constructor needs two parameters: the file descriptor for mom (it has just been opened) and the name that was typed on the command line.

When run() returns, close the socket and terminate.

Creating the Kid processes. Your program should work if the Kid processes are on a remote machine or on the localhost, or a mixture. The major difference will be timing: anything running over the internet will be slower. It is adequate to test your code using localhost, but more fun to run it with a pile of kids on a remote machine.

Here is a tcsh shell script to start up any number of kids you like. It uses the file named "kids.txt", which contains names for the kids. The shell script uses however many names are in the file. Your mom code is supposed to stop welcoming new kids after 25 have arrived. If she closes the welcome socket, any additional arrivals will be refused and nothing bad will happen.

```
#!/bin/tcsh -fv

if ($# != 1) then
    echo "usage: $0 host"
    exit 1;
```

```
endif

foreach x ( `cat kids.txt` )
    kid $1 $x >! $x.log  &
end
```

5.1 A Kid's Logic

As the game progresses through stages of the protocol, a kid must remember what stage he has reached in order to be able to validate the next protocol message. For this purpose, I recommend using an enum type for stages and an enum variable in each kid to record the current stage:

```
enum StateT { NEWBIE, MARCHING, SEEKING, SITTING, QUIT };
```

The kid must check and change this variable in the doAction() function as he moves from one stage to the next, ending with QUIT, which is the signal to terminate.

- Kid class members. The Kid will need several class members to store private data:

  ```
  int momFd;
  FILE* momIn, * momOut;
  int nChairs;
  string kidName;
  StateT pcol = NEWBIE;
  ```

- `run()`
  Open input and output streams on the socket and execute commands until the kid's state is set to QUIT.
- `doCommand()`
  Read a message (print it for debugging) and, depending on the command part of the message, execute one of the six doAction functions. Most of these functions change the kid's state variable.

  – `doGetup()`: Download the current number of chairs from the message and store it.
  – `doSit()`: Send a chair request to mom.
  – `doAck()`: Print a contented message on the screen and wait for the next command to come in.
  – `doQuit()`: Print a sad message on the screen and wait for the run() function to terminate you.
  – `doPrize()`: Print a happy message on the screen and wait for the run() function to terminate you.
  – `doNack()`: Download the list of available seats, find one that is available, and request it. (Hint: you can use strchr() to find a seat.)

6 Implementation Notes.

Time is short and this is not an easy assignment. For guidance, I am providing several code fragments and the `doCommand()` function, below.

- Use a random wait in the doSit() function. usleep(rand(30000)); Otherwise, the last kids in the list will always lose and the first one will almost always win. In addition, my implementation uses randomization in the order in which Mom tells the kids to GETUP and to SIT. You don't need to go that far to make a fair game; ask if you want me to send you the code.

- This function gives a single point to catch errors of all sorts identified by the action functions.

```
//-----------------------------------------
void KidLogic::doCommand(){
    int status = fscanf( momIn, "%6s", command );
    if (status!=1) fatalp("Error reading command");
    cout << "State = "<< stateName[pcol]<< ",
            Command is: " <<command << endl ;
    try{
        if (strcmp( "GETUP", command) == 0) doGetup();
        else if (strcmp( "SIT", command) == 0) doSit();
        else if (strcmp( "NACK", command) == 0) doNack();
        else if (strcmp( "ACK", command) == 0) doAck();
        else if (strcmp( "QUIT", command) == 0) doQuit();
        else if (strcmp( "PRIZE", command) == 0) doPrize();
        else throw( "Protocol is mixed up." );
    }
    catch (string& s) {
        cout << s <<" ["<< command <<"]\n";
        exit(1);
    }
}
```

  The doCommand() function dispatches the work to one of six action functions. The action functions are all very targeted and short. The functions for doGetup and doNack need to read the rest of the command from the instream. The biggest benefit of using the stream interface is that reading the rest of the command is easy.

- I am using strchr in the doNack function to find a chair that has not already been taken.

7 Submission

Turn in all the code and header files, your data file if you have one, and the screen output:

- The first run should have three kids. Copy the screen output from Mom and each Kid into a single file, with suitable comments saying which process produced each set of output. Make sure that Mom's output shows the sequence of commands that have happened, and which kid was involved in each command. Make sure that every command in the protocol shows up.

- The second run should have many children. Turn in Mom's screen output.