Episode 7 (How Chat Works II)

Tanuki! You know that chat is real-time textual intercommunication. You know that in order to communicate with each other, you have to have a mutual process (a running program). It's like talking to each other with mutual smartphones. The phones can be from the same carrier (think of it as a retailer) or from different carriers. But the mechanism is the same for both parties. You just switch to the sender and the receiver. In the same way, the server program and the client program of a chat program are almost the same. So, since repeating the same explanation over and over again will only bore you, I'll only explain the part about switching between sending and receiving!

OK, OK, I'm sick and tired of having the same things explained to me over and over again.

Now, I will present a client program (cclient.c) that asks the server program. Note that the header file (professionally created program) (netdb.h), which is read by the program, defines a set of variables (structure) necessary for communication.

Client program (cclient.c) ---------1

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#define PORT     (in_port_t)50000 /* Server (self) port number */
#define BUF_LEN  512              /* Buffer size for sending/receiving */

main()
{
        /* Variable Declaration */
        struct hostent  *server_ent; /* Server (counterparty) information */
        struct sockaddr_in  server;  /* Server (opposite) address */
        int soc;                     /* Socket descriptor */
        char hostname[]="Server Name";  /* Server (peer) host name: localhost, etc. */
                                     /* IP address of the server is also acceptable */
        char buf[BUF_LEN];           /* Transmit/receive buffer */

        /* Obtain address information from the server's (the other party's) host name */
        if((server_ent = gethostbyname(hostname)) == NULL){
                perror("gethostbyname");
                exit(1);
        }

        /* Stores the server (peer) address in the sockaddr_in structure  */
        memset((char *)&server, 0, sizeof(server));
        server.sin_family = AF_INET;
        server.sin_port = htons(PORT);
        memcpy((char *)&server.sin_addr, server_ent->h_addr,
               server_ent->h_length);

        /* Create stream-type socket with IPv4  */
        if((soc = socket(AF_INET, SOCK_STREAM, 0)) < 0){
                perror("socket");
                exit(1);
        }
```

Client program (cclient.c) ---------2

```
        /* Connect to server (other party)  */
        if(connect(soc, (struct sockaddr *)&server, sizeof(server)) == -1){
                perror("connect");
                exit(1);
        }

        /* the other party is first */
        write(1, "Wait¥n", 5);

        /* Communication loops */
        do{
                int n;                      /* Bytes read  */

                n = read(soc, buf, BUF_LEN); /* Read from socket soc  */
                write(1, buf, n);            /* Write to standard output 1  */
                n = read(0, buf, BUF_LEN);   /* Read from standard input 0  */
                write(soc, buf, n);          /* Export to socket soc */
        }
        while( strncmp(buf, "quit" ,4) != 0 );        /* end decision */

        /* Closing the socket */
        close(soc);
}
```

structure hostent (netdb.h)

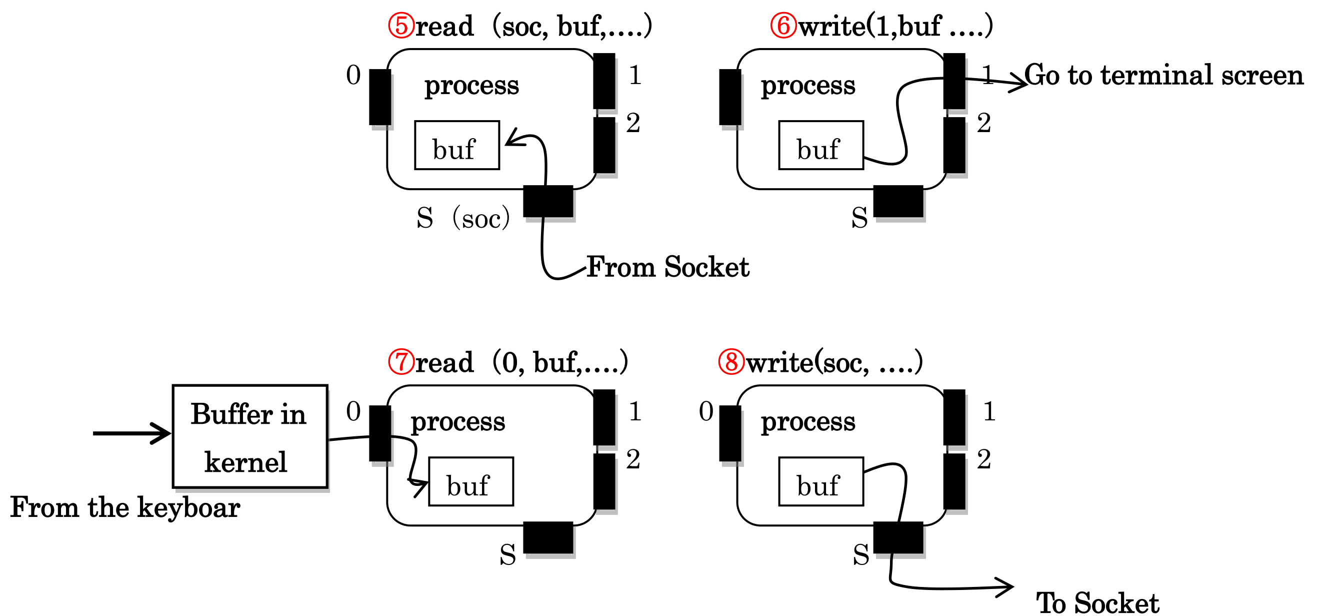
```
struct hostent {
    char  *h_name;             /* official name of host */
    char **h_aliases;          /* alias list */
    int    h_addrtype;         /* host address type */
    int    h_length;           /* length of address */
    char **h_addr_list;        /* list of addresses */
#define h_addr h_addr_list[0]  /* h_addr is the first address in h_addr_list.*/
                               /* For compatibility with the past */
}
```

Kitsune! I don't know anything about structures and such. ・・・・!

That's right, the hardest part of C to understand is pointer variables (variable names with an asterisk (*)) and structures. But you can create pseudo-classes with structs and pointer variables. In other words, it is an important part of C because it is connected to the basic idea of classes in object-oriented languages, but it may be impossible without a thorough study of the C language. I should give up... If I give up, I might not be able to live in the animal world.

Even if you don't know C, you can at least understand how it works because it illustrates the flow of data (messages).
With the read instruction in ⑤, a message from the server comes through the socket into the buffer (buf) of the client process. The write instruction ⑥ displays the message from the buffer (buf) on the specified monitor.
The read instruction in ⑦ brings the reply message for the server input from the keyboard on the client side into the buffer (buf) of the process, and the write instruction in ⑧ brings it from the file descriptor (S) assigned to it through the socket to the server process.
Even a raccoon can figure this out. Also, don't you think it is amazing that a computer can create both such a program and such a mechanism?

Data flow within the client process



Kitsune! Tell me about the connect() function on line 38 of the program!

The connect() function checks to see if the client process has established a connection with the server process.
In other words, as shown in ⑤ and ⑧ above, the socket descriptor (S) is the entry/exit point, and the function determines whether inter-process communication is established through the socket.
If the connection is established and communication is possible, the return value of the function is 0.
1 means that no matter how hard you try, communication is not possible.
The details are as follows!

Details of connect() function

Connect to the IP address that references (destinations) the socket (descriptor).
Once the connection is established, read and write operations from the socket descriptor are possible.

connect (Socket descriptor, destination IP address, length of IP address (4 bytes))

```
connect(soc, (struct sockaddr *)&server, sizeof(server))
```

Return value: 0 for success (establishment), -1 for failure

There are a few C functions in the program, so I'll explain them to you so that you can save time looking them up!

Review of C language 1 Arrow operator

```
server_ent->h_addr
```

Copies the received server information (pointer variable) to a member (h_addr) of the client's structure (hostent).

server_ent : Pointer variable of structure hostent (must be a pointer variable)

-> : Arrow operator

h_addr : Member of structure (hostent)

Review of C language 2 Memory initialization

```
memset((char *)&server, 0, sizeof(server))
```

The length of the structure is initialized to 0 from the beginning of the structure server (pointer variable) cast to the string to the length of the structure.

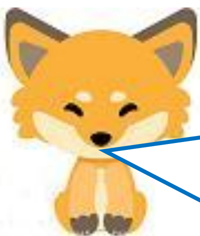
Review of C language 3 Data Copy

copy destination copy source Number of bytes to copy

```
memcpy((char *)&server.sin_addr, server_ent->h_addr, server_ent->h_length)
```

The received server IP address (h_addr) is copied into the client structure sockaddr_in member (sin_addr) of the client structure sockaddr_in for the bytes specified by h_length.

Kitsune, I entered the program and saved it, but what do I do after that?

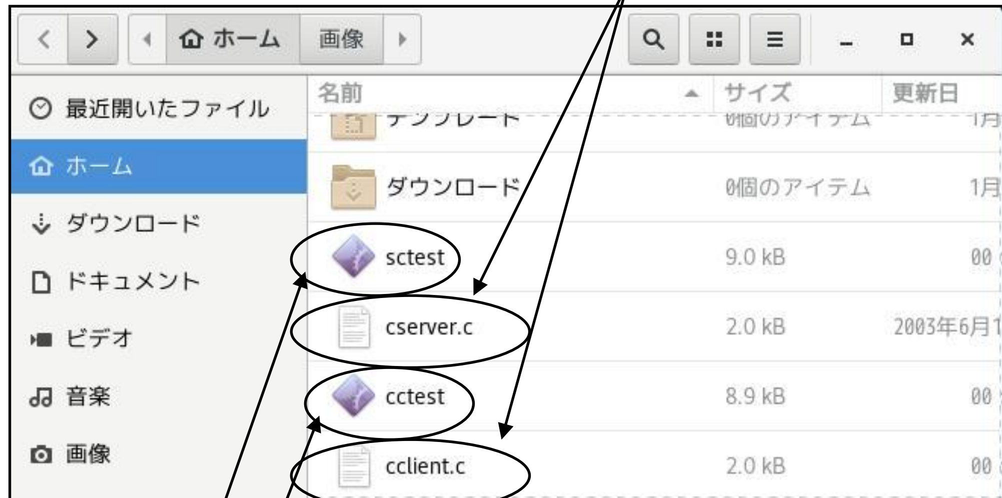


I will compile the saved server and client programs, run them, and run the two processes on one OS (CentOS), which may be boring if you don't have to chat on two PCs, but it is easier to bug out the programs if you run them on one OS to make sure they work. It is easier to bug fix the program if you run it on one OS.

Next, I will show you how to run the program.

Compilation of chat programs

①Save the created source files (cserver.c and cclient.c) in your home directory.



②Compile. The executable files are assumed to be "sctest" and "cctest" respectively.

Create server executable

```
fox @localhost~:$ gcc –o sctest cserver.c
```

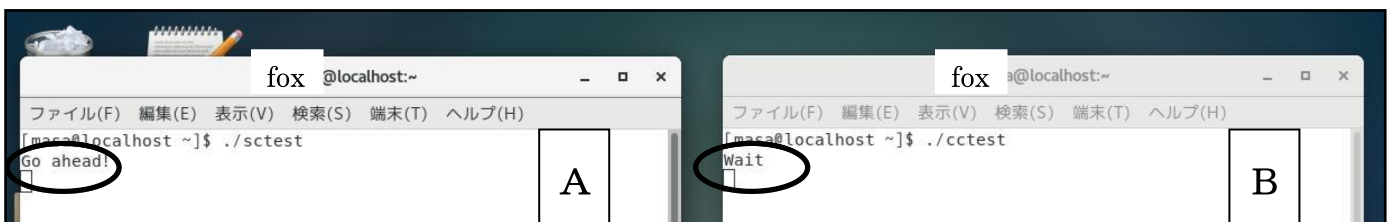
Create client executable

```
fox @localhost~:$ gcc –o cctest cclient.c
```

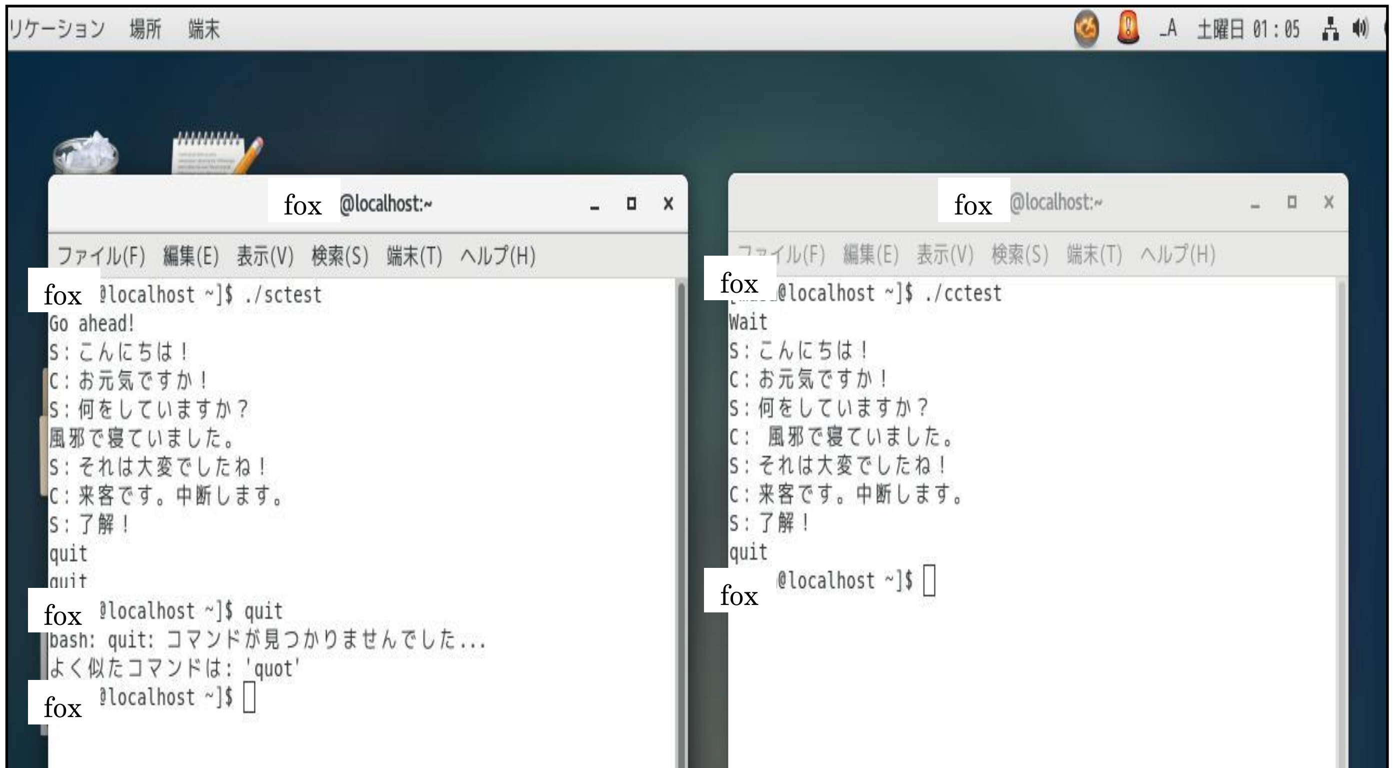
Running the chat program-1

①When running a server process and a client process on a single PC (host)

- Start two terminals. The one that starts the server process is [A] and the one that starts the client process is [B].
- Activate [A] first, type ". /sctest" and enter the waiting state.
- Then activate [B] and type ". /cctest". The next moment, the message "Go ahead" (I'm ahead) appears in [A] and "Wait" (I'll wait) in [B].is ahead" in [A] and "Wait" in [B].



②Start the chat by repeating the order [A] first, then [B]. If you make a mistake in the procedure, the system will not respond correctly.



③To end, type "quit".

Once the server and client programs are fully operational, the next step is to start chatting between the two PCs!
If you can't chat, there is nothing wrong with the program, so you can isolate the problem to the communication cable or IP address settings.

Running the chat program-2

①When running one server process and multiple client processes on two or more PCs (hosts)

- Start a terminal on the PC that will start the server process. Let it be the server [A].
- Start the terminal that will launch the client process. Let it be Client [B].

  [IMPORTANT] You need to modify and compile the client program before starting it up!

```
char hostname[]="Set the IP address of the server to request a connection";
```

- Activate [A] first, type ". /sctest" and enter the waiting state.
- Then activate [B] and type ". /cctest". The next moment, the message "Go ahead" (I'm ahead) will appear in [A] and "Wait" (I'll wait) in [B].
- The rest is the same as in Running the chat program-1.

Interesting, interesting!
I feel a sense of accomplishment not only in executing a program created by others as an application, but also in chatting with people after understanding how it works. It was a lot of work to get to this point, but it's very satisfying. Thank you, Kitsune.

You're welcome. I'm glad the raccoon dog likes you too.

Tanuki! Let's take a break,

Let's move on to Episode 8!

Translated at DeepL