Episode 6 (How Chat Works I)

Tanuki, it's time to learn a computer language. As I mentioned before, the language that computers can understand is machine language, which is input in binary (or hexadecimal). The closest language to machine language is assembler, but that is determined by the specifications of the CPU. In contrast, there are BASIC, Python, Scratch, etc., which are called interpreted languages. An interpreted language is one in which each instruction is converted and executed in machine language. There are also compiler languages, which cannot be executed until the entire source program has been converted to machine language. C, Java, C++, Cobol, and Fortran belong to these compiler languages. There are also languages called PHP, JSP, and ASP, which are used for server-side programming. These are as easy to use (in the sense that they do not need to be compiled) as an interpreter. Furthermore, the above languages are divided into procedural languages such as BASIC, C, Cobol, and Fortran, and object-oriented languages such as Java, C++, Python, PHP, JSP, and ASP. Except for assembler, the other languages are also called high-level languages because their codes (instructions) are easy to understand by humans because they use letters similar to English words. As before, the chat program will be written in C, which is both a procedural language and a compiler language. The reason is that it is easy to imagine the operation to hardware and files.

I know what you mean. Object-oriented languages have a lot of black boxes called classes, so I don't really understand what they are doing.

Yes, Tanuki, you can make a chat program in Java, but I don't see the socket part, etc. I may cover the Java language in the near future!
Well, let's get on with the chat program, shall we?

We will create two chat programs, one as a server program (cserver.c) and the other as a client program (cclient.c). Both programs are similar, but there are some differences. The server is the service provider and the client is the customer receiving the service. I will first present the server program and explain its contents.
I won't go into the details of the C language, so you'll have to learn it by yourself. If you can't read it, you'll have to start over from studying Japanese.

Tanuki, please remind me of the steps to run the C language, which I explained last time. First, start CentOS, type in (cserver.c) with gedit, save it to the current directory, compile it, and run it to get the process.

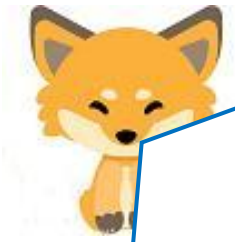
Server program (cserver.c) ---------1

```
#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 sockaddr_in me; /* Define sockaddr_in structure with me    */
        int soc_waiting;       /* Socket to listen on   */
        int soc;               /* Sockets used for sending and receiving */
        char buf[BUF_LEN];     /* Transmit/receive buffer   */
        /* Stores the server (self) address in the sockaddr_in structure   */
        memset((char *)&me, 0, sizeof(me));  /* Initialize the sockaddr_in
                                                structure with 0   */
        me.sin_family = AF_INET; /* IPv4 */
        me.sin_addr.s_addr = htonl(INADDR_ANY);  /* INADDR_ANY is an IP
                                                    address     */
        me.sin_port = htons(PORT);  /* PORT=50000: 16-bit integer    */
        /* Create stream-type socket with IPv4   */
        if((soc_waiting = socket(AF_INET, SOCK_STREAM, 0)) < 0 ){
                perror("socket");
                exit(1);
        }
        /* Set the server (self) address to the socket */
        if(bind(soc_waiting, (struct sockaddr *)&me, sizeof(me)) == -1){
          perror("bind");
          exit(1);
        }
        /* Set to listen on socket     */
        listen(soc_waiting, 1);
 /* Connection request established and new socket descriptor returned   */
        soc = accept(soc_waiting, NULL, NULL);
        /* Close socket descriptor for connection waiting   */
        close(soc_waiting);
        /* Click here first   */
        write(1, "Go ahead!¥n", 10);
        /* Communication loops   */
```

Server program (cserver.c) --------2

```
        do{
          int n;                        /* Bytes read  */
          n = read(0, buf, BUF_LEN);    /* ①Read from standard input 0 */
          write(soc, buf, n);           /* ②Export to socket soc */
          n = read(soc, buf, BUF_LEN); /* ③Read from socket soc */
          write(1, buf, n);             /* ④Write to standard output 1 */
        }
        while (strncmp(buf,"quit",4) != 0);         /* end decision */
        /* Close socket descriptor for communication */
        close(soc);
}
```

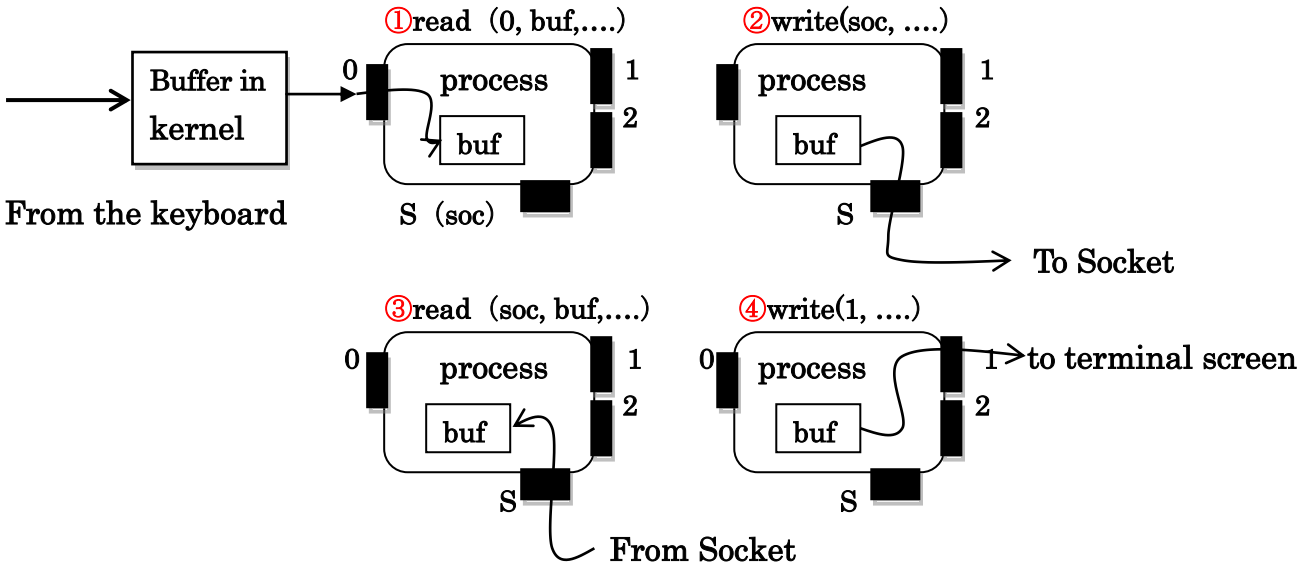


Tanuki, let me explain a little about the flow of data after execution.

When you type "Hello! from the keyboard, the message goes through the buffer in the kernel (base of the OS) and is stored in a buf (buffer) in the process from the entrance of the standard input (file descriptor (FD): 0) with the read instruction in ①.

Next, the message goes out from the buf in the process to the socket in the OS with the FD allocated as the exit (S) by the write instruction in ②. Conversely, the message from the other party arrives from the socket in the OS through the allocated FD (S) to the buf in the process (③), and is displayed on the terminal screen (monitor) by leaving standard output 1 with the write instruction. That is ④.

Data flow within the server process



Fox, on line 22 of the program. socket(AF_INET, SOCK_STREAM, 0), which defines the specification of the socket to be created in the OS, I think.

Tanuki, good on you for noticing!
Okay, I'll summarize how it works below.

SOCKET() function details

protocol family	communication system	Protocol to be used
socket(AF_INET ,	SOCK_STREAM,	0)
communication domain (IPv4)	When connection type	IPPROTO_TCP or 0
AF_INET6 (IPv6)	When datagram type	IPPROTO_UDP
AF_UNIX (local communication)	(SOCK_DGRAM)	
AF(Abbreviation for Address Family)		

Return value : If the connection is successful. Assigned file descriptors (>2)
If the connection fails. -1 (<0)
soc_waiting = return value (one of 3,4,5・・・・) Let's check it!

Kitsune, while you're at it, tell me htonl(INADDR_ANY) on line 19.

OK, INADDR_ANY contains an IP address such as "192.168.0.5". The INADDR_ANY function tells the memory to store this IP address as "5.0.168.192" or "192.168.0.5" for each byte (8 bits). When the data is sent over the Internet, including the network, it is sent in the order of all 5s, then 0s, then 168s, and finally 192s. Of course, this is not done in decimal, but converted to binary numbers of 0s and 1s.

htonl() function (Abbreviation for host to network long (32-bit))

A function that performs 32-bit byte ordering (Big-endian method). Byte-order means that the larger part of a 32-bit integer is stored in the smaller address. When sending data over the Internet, data is sent out in order from the smallest address to avoid confusion with the other party and to keep synchronization.

・Intel CPUs are stored in memory in a Little-endian fashion.
Intel (Little-endian method) Also called host byte order!

(A+3) address	(A+2) address	(A+1) address	A address
DD	CC	BB	AA

・When sending packets to the Internet, they are sent using the Big-endian method.
Big-endian method Also called network byte order!

(A+3) address	(A+2) address	(A+1) address	A address
AA	BB	CC	DD

→ Transmission over the Internet

How can I check whether the CPU of my computer belongs to the host byte order or the network byte order?

You can find out by running the following program (bite.c) on the raccoon dog's computer. If the first line of the program displays [DD:CC:BB:AA], the raccoon dog's CPU is in host byte order. Conversely, if the order [AA:BB:CC:DD] is displayed, it is the network byte order.

Confirmation program (bite.c) : Let's check it out!

```
#include <stdio.h>

int main() {
        unsigned int iVal = 0xAABBCCDD; /* hex 32-bit */
        unsigned char *pc = (char *)&iVal;
        /* Save as is Little-endian method (Intel) */
        printf("%X : %X : %X : %X¥n", pc[0], pc[1], pc[2], pc[3]);
      /* Converted to Big-endian method  */
        iVal = htonl(iVal); /* Abbreviation for host to network long (32-bit) */
        printf("%X : %X : %X : %X¥n", pc[0], pc[1], pc[2], pc[3]);
      /* Return to Little-endian method */
        iVal = ntohl(iVal);
        printf("%X : %X : %X : %X¥n", pc[0], pc[1], pc[2], pc[3]);
        return 0;
}
```

Result

```
Fox@cent2020:~
ファイル(F)  編集(E)  表示(V)  検索(S)  端末(T)  ヘルプ(H)
Fox@cent2020 ~]$ gcc -o bitetest bite.c
Fox@cent2020 ~]$ ./bitetest
DD : CC : BB : AA    Host Byte Order
AA : BB : CC : DD    Network Byte Order
DD : CC : BB : AA    Host Byte Order
Fox@cent2020 ~]$
```

By the way, I'll add the style of the bind() function in line 27, which sets the IP address set in the raccoon dog's PC to the socket, below.

Details of BIND() function

file descriptor　　pointer of address structure　　length of address structure

```
bind( soc_waiting , (struct sockaddr *)&me , sizeof(me) )
```

Return value of bind() : Success in naming the socket: 0 Failure: -1

Contents of address structure (sockaddr_in) in case of AF_INET (IPv4)

Defined in #include<netinet/in.h>.

```
struct sockaddr_in{
        uint8_t         sin_len;
        sa_familiy_t    sin_family;  /*Specification of AF_INET (IPv4)*/
        in_port_t       sin_port;    /*Server port number (50000)*/
        struct in_addr  sin_addr;    /* Server IP Address*/
        char            sin_zero[8];
};
```

And while you're at it, explain the listen() and accept() functions.

The listen() function is the function that is always listening for connection requests from client processes. Needless to say, this is to send a message to the server. When a connection request is received, the accept() function allocates one socket (file) descriptor (positive numbers from 0 to 1023) and establishes a connection. In this case, it prepares a new entrance to accept messages from the client process.

Details of the listen() function

listen (file (here socket) descriptor, 1)

1 : It means one connection request to wait.

The Listen function is used in a connected network where one server (process) waits for connection requests from multiple client (process) sockets.

Return value: 0 for success, -1 for failure.

Details of accept() function

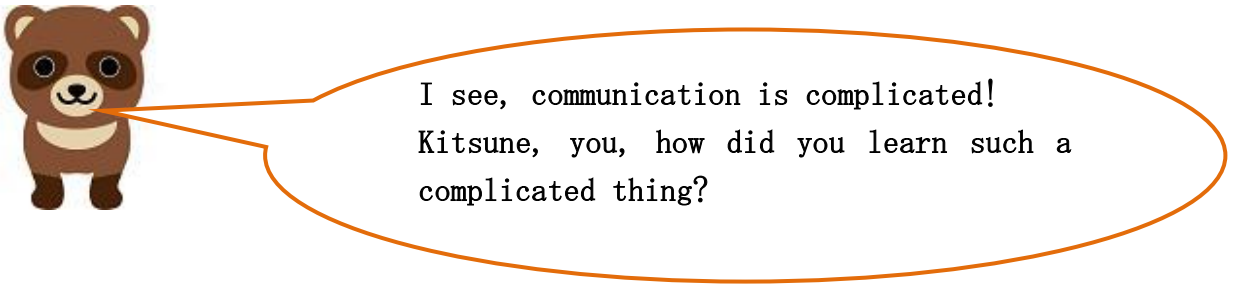
accept(file descriptor, port number and address of structure sockaddr, length of structure)

file descriptor : The socket descriptor used for the connection.

Port number and address of structure sockaddr : Information (IP address, etc.) of the client (process) requesting the connection.
When there is no need to return information, specify NULL.

Structure Length : Bit length of the entire structure (information).
If the second argument is null, null is specified.

return value : If successful, integer value of the new socket descriptor. If failed, -1.
The cause of the failure is that the first argument is not a file (socket) descriptor, and the socket() function failed to create the socket.



About 40 years ago, when 8-bit computers were just starting to appear on the market, there was no Internet system, so I had no choice but to read books.
That's why reading comprehension in Japanese is so important. Nowadays, we are lucky because we can learn almost anything by searching on the Internet. However, it is now necessary to be able to judge whether a statement is correct or incorrect.

I still think it would be better to study english,mathematics, japanese, science, and social studies (especially history) well in high school.

I mean to acquire the ability to make comprehensive judgments. I am in the business of making dummies, so I will be making a scientific guise for my dummies.
Read with skepticism.

Tanuki! Let's take a break to rest our heads.

Next up, Episode 7.

Translated at DeepL