# 第６話（チャットの仕組みＩ）

タヌキ、そろそろコンピュータ言語の勉強をしましょうか。前にも言ったけれども、コンピュータが理解できる言語は２進数（あるいは１６進数）で入力されるマシン（機械）語なのだ。マシン語に一番近い言語はアセンブラ言語なのだが、それがＣＰＵの仕様によって決まるのだ。それに対してインタプリタ言語と言われる BASIC 言語、Python、スクラッチなどがあります。インタプリタ言語とは、命令毎にマシン語に変換、実行される言語です。さらにソースプログラム全体をマシン語に変換してからでないと実行できないコンパイラ言語があります。このコンパイラ言語に属するのが C 言語、Java、C++、Cobol、Fortran なのだ。また、サーバ側のプログラミングに使われる PHP、JSP、ASP などという言語があります。これはインタプリタと同じくらい手軽（コンパイルする必要は無いという意味で）に使えます。さらに、上記の言語は、BASIC、C 言語、Cobol、Fortran などの手続き型言語と Java、C++、Python、PHP、JSP、ASP などのオブジェクト指向言語に分かれるんだ。アセンブラ以外は、コード（命令）が英語の単語に近い文字が使われているので、人間にわかりやすい、ということで高級言語とも言われています。チャットのプログラムは前回同様、手続き型言語であり、コンパイラ言語でもあるＣ言語を用いて作ります。理由はハードやファイルへの動作がイメージし易いからです。

オイラも知っているよ。オブジェクト指向言語ってクラスというブラックボックスの部分が多くて何をやっているのか良くわからないんだよね。

そうなんだタヌキ、Java でもチャットプログラムは作れるけれども、ソケットの部分などが見えてこないんだ。
Java 言語については、そのうちに取り上げることもあるよ！
じゃあ、チャットのプログラムに取り掛かろうか。

チャットプログラムは、２個作成し、１台のＰＣ（CentOS）の中で２個起動し、コミュニケーションすることになります。１つはサーバプログラム（cserver.c）とし、もう１つはクライアントプログラム（cclient.c）とするよ。両方とも似たようなプログラムだけれども、若干違うところがあるよ。サーバとはサービスを提供し、クライアントはサービスを受ける顧客という意味ですよ。
最初にサーバプログラムを提示し、内容の説明をするね。
サーバプログラムをコンパイルし、実行するとプロセスＡとなることを覚えておいてください。Ｃ言語の詳細な説明はしないから、タヌキ、自分で勉強してください。Ｃ言語の解説本ぐらい読めるだろ。読めなければ、国語の勉強からやり直しだな。

タヌキ、前回説明した、Ｃ言語の実行までの手順を思い出してくれよ。まず、CentOS を起動し、（cserver.c）を gedit で入力し、カレントディレクトリに保存し、コンパイルし、実行するとプロセスになるよ。

サーバプログラム（cserver.c）---------1

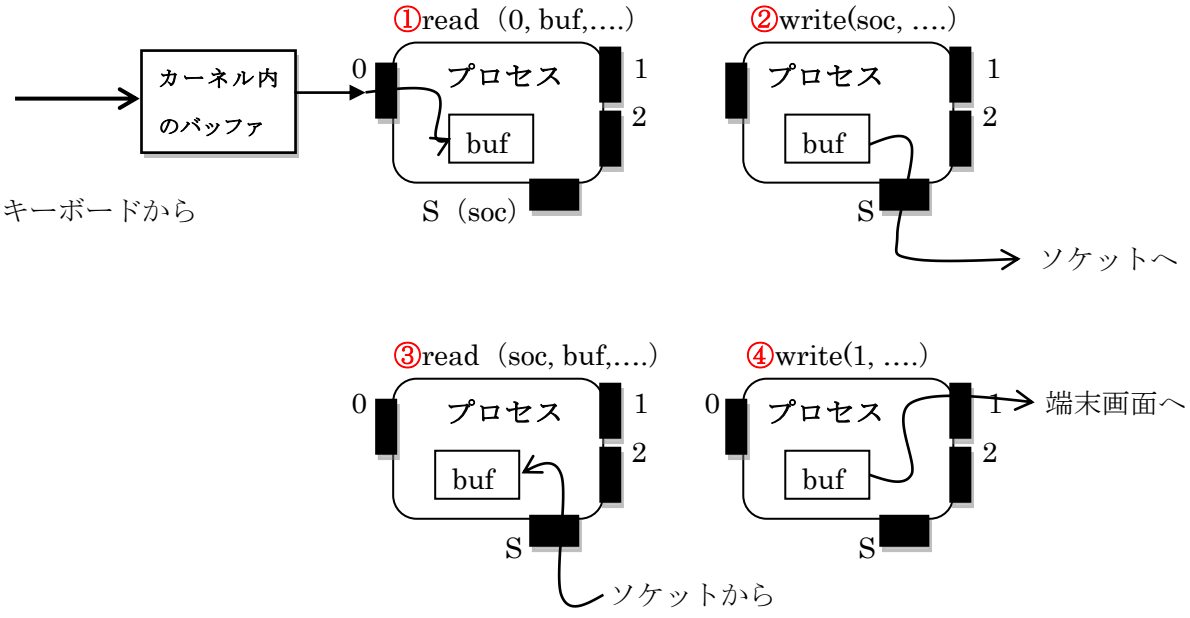
```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#define PORT     (in_port_t)50000 /* サーバ(自分)のポート番号 */
#define BUF_LEN  512              /* 送受信のバッファの大きさ */

main()
{
        /* 変数宣言 */
        struct sockaddr_in me; /* sockaddr_in 構造体を me で定義    */
        int soc_waiting;       /* listen するソケット    */
        int soc;               /* 送受信に使うソケット */
        char buf[BUF_LEN];     /* 送受信のバッファ      */
        /* サーバ(自分)のアドレスを sockaddr_in 構造体に格納  */
        memset((char *)&me, 0, sizeof(me));  /* sockaddr_in 構造体を 0 で初期化*/
        me.sin_family = AF_INET; /* IPv4 */
        me.sin_addr.s_addr = htonl(INADDR_ANY);  /*INADDR_ANY は IP アドレス */
        me.sin_port = htons(PORT);  /*PORT=50000:16 ビットの整数*/
        /* IPv4 でストリーム型のソケットを作成  */
        if((soc_waiting = socket(AF_INET, SOCK_STREAM, 0)) < 0 ){
                perror("socket");
                exit(1);
        }
        /* サーバ(自分)のアドレスをソケットに設定 */
        if(bind(soc_waiting, (struct sockaddr *)&me, sizeof(me)) == -1){
          perror("bind");
          exit(1);
        }
        /* ソケットで待ち受けることを設定 */
        listen(soc_waiting, 1);
        /* 接続要求が確立し、新しいソケットディスクリプタを返す */
        soc = accept(soc_waiting, NULL, NULL);
        /* 接続待ちのためのソケットディスクリプタを閉じる */
        close(soc_waiting);
        /* こちらから先 */
        write(1, "Go ahead!¥n", 10);
        /* 通信のループ */
```

```
        do{
          int n;                          /* 読み込まれたバイト数  */
          n = read(0, buf, BUF_LEN);      /* ①標準入力 0 から読む      */
          write(soc, buf, n);             /* ②ソケット soc に書き出す */
          n = read(soc, buf, BUF_LEN);    /* ③ソケット soc から読む    */
          write(1, buf, n);               /* ④標準出力 1 に書き出す    */
        }
        while (strncmp(buf,"quit",4) != 0);          /* 終了判定 */
        /* 通信用のソケットディスクリプタを閉じる */
        close(soc);
}
```

タヌキ、実行後のデータの流れをチョット説明しとくね。
キーボードから「こんにちは！」と入力すると、そのメッセージはカーネル内（ＯＳの基）のバッファを通って①の read 命令で標準入力（ファイルディスクリプタ（FD）：0）の入り口からプロセス内の buf（バッファ）に保存されます。次に②の write 命令でプロセス内の buf から割り当てられた FD を出口（S）としてメッセージが、OS 内のソケットに出て行きます。逆に通信相手からのメッセージは、OS 内のソケットから、割り当てられた FD（S）を通ってプロセス内の buf に届き（③）、write 命令で標準出力 1 から出て、端末画面（モニター）に表示されます。それが④です。

サーバプロセス内のデータの流れ



キツネ、プログラムの 22 行目に socket(AF_INET, SOCK_STREAM, 0)という関数があり、これが OS 内に作られるソケットの仕様を定義している、と思うんだが、その内容を教えてくれ

タヌキ、良く気がついたな！
じゃあ、以下にその働きをまとめておくね。

SOCKET()関数の詳細

| | プロトコルファミリー | 通信方式 | 使用するプロトコル |
|---|---|---|---|
| socket( | AF_INET , | SOCK_STREAM, | 0 ) |
| | 通信ドメイン（IPｖ4） | コネクション型　の時 | IPPROTO_TCP あるいは 0 |
| | AF_INET6（IPｖ6） | データグラム型　の時 | IPPROTO_UDP |
| | AF_UNIX（ローカル通信） | (SOCK_DGRAM) | |
| | AF(Address Family の略) | | |

返り値：接続に成功した場合。割り振られたファイルディスクリプタ（＞２）
接続に失敗した場合。－１（＜０）
soc_waiting = 返り値（3,4,5・・・・のどれか）　確認してみよう！

キツネ、ついでに 19 行目の htonl(INADDR_ANY)も教えてくれ。

OK、INADDR_ANY には、「192.168.0.5」のような IP アドレスが格納されているんだ。この IP アドレスを 1 バイト（8 ビット）毎に「5.0.168.192」とメモリに記憶させるか、「192.168.0.5」と記憶させるか指示する関数なのだ。ネットワークを含めてインターネット上に送信される時には、全て 5、次に 0、次が 168、最後に 192 という順で送信されるのだがね。当然 10 進数ではなく、0、1 の 2 進数に変換されてのことだよ。

**htonl()関数**　(host to network long(32 ビット)の略)

**32 ビットのバイトオーダー（Big-endian 方式）を行う関数。バイトオーダーとは、32 ビットの整数の大きい部分を小さい番地に格納する。ネット上に送出する時に、小さい番地から順にデータを送出し、相手との混乱を避け、同期を取っている。**

**・インテルＣＰＵはLittle-endian 方式でメモリーに格納される。**

**インテル（Little-endian 方式）　ホストバイトオーダーともいう！**

| （A+3）番地 | （A+2）番地 | （A+1）番地 | A 番地 |
|---|---|---|---|
| DD | CC | BB | AA |

**・インターネットにパケットを送信する時は、Big-endian 方式で送信される。**

**Big-endian 方式　ネットワークバイトオーダーともいう！**

| （A+3）番地 | （A+2）番地 | （A+1）番地 | A 番地 |
|---|---|---|---|
| AA | BB | CC | DD |

→　ネット上に送信

オイラのパソコンの CPU が**ホストバイトオーダーかネットワークバイトオーダーのどちらに属するのか確認するにはどうするの？**

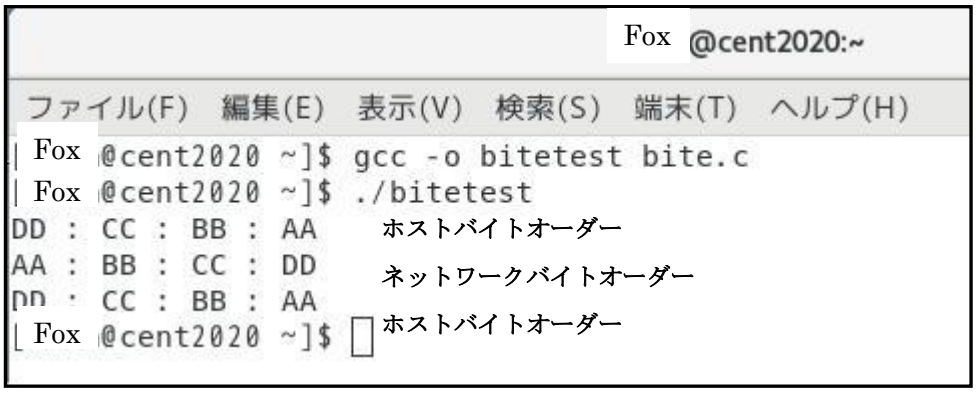
タヌキのパソコンで次のようなプログラム（bite.c）を実行してみればわかるよ。実行した時に、最初の１行目で［DD : CC : BB : AA］と表示されたらタヌキの CPU はホストバイトオーダーだ。逆に［AA : BB : CC : DD］という順に表示されたらネットワークバイトオーダーだ。

**確認プログラム（bite.c）：確認してみよう！**

```
#include <stdio.h>

int main() {
        unsigned int iVal = 0xAABBCCDD; /* 16進32ビット */
        unsigned char *pc = (char *)&iVal;
        /* そのまま保存  Little-endian方式（インテル）*/
        printf("%X : %X : %X : %X¥n", pc[0], pc[1], pc[2], pc[3]);
      /* Big-endian方式に変換 */
        iVal = htonl(iVal);   /* host to network long(32ビット）の略 */
        printf("%X : %X : %X : %X¥n", pc[0], pc[1], pc[2], pc[3]);
      /* Little-endian方式に戻す */
        iVal = ntohl(iVal);
        printf("%X : %X : %X : %X¥n", pc[0], pc[1], pc[2], pc[3]);
        return 0;
}
```

**結果**



ちなみにタヌキの PC に設定されている IP アドレスをソケットに設定する 27 行目の bind()関数の様式を以下に追加しておくよ。

BIND()関数の詳細

ファイルディスクリプタ　　アドレス構造体へのポインタ　　アドレス構造体の長さ

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

bind（）の返り値：　ソケットに名前を付けるのに成功：0　失敗：－1

AF_INET（IPｖ4）の場合のアドレス構造体（sockaddr_in）の内容

(#include<netinet/in.h>で定義されている)

```
struct sockaddr_in{
        uint8_t          sin_len;
        sa_familiy_t     sin_family;  /*  AF_INET（IPｖ4）の指定
        in_port_t        sin_port;  /*  サーバのポート番号（50000）*/
        struct in_addr   sin_addr;  /*  サーバの IP アドレス  */
        char             sin_zero[8];
    };
```

ついでに、listen()関数と accept()関数についても説明してよ。

listen()関数は、クライアントプロセスからの接続要求がこないか常に耳を傾けている関数なのだ。言うまでも無くサーバにメッセージを送信する為です。接続要求があると、ソケット（ファイル）ディスクリプタ（正数値 0 から 1023 まで）を 1 個割り当てて接続を確立する関数が accept()関数だ。この場合、クライアントプロセスからのメッセージを受け入れる為の新しい入り口を用意することになる。

**listen()関数の詳細**

**listen（ファイル（ここではソケット）ディスクリプタ、１）**

**１：待ち受ける接続要求は１個を意味している。**

Listen 関数は、１つのサーバ（プロセス）が複数のクライアント（プロセス）のソケットからの接続要求を待つコネクト型のネットワークで使われる。

返り値：成功は 0、失敗は－１

**accept()関数の詳細**

accept(**ファイルディスクリプタ, 構造体 sockaddr のポート番号、アドレス, 構造体の長さ**)

ファイルディスクリプタ：接続に使われるソケットディスクリプタのこと。

**構造体 sockaddr のポート番号、アドレス**：接続要求してくるクライアント（プロセス）の情報（IP アドレス等）。**情報を返す必要が無い時は NULL を指定する。**

**構造体の長さ**：構造体（情報）全体のビット長。**第二引数が NULL の時は NULL を指定する。**

返り値：**成功した場合は、新しいソケットディスクリプタの整数値**。失敗した場合は－１。

失敗の原因は、第一引数が、ファイル（ソケット）ディスクリプタになっていない、socket()関数によるソケット作成の失敗。

なるほど、通信てのは複雑なんだ！
キツネ、おめえ、こんな複雑なこと
どうやって勉強したんだ。

独学だよ。４０年程前は８ビットのパソコンが市中に出回り始めた頃で当然インターネットというシステムもなかったから、ひたすら本を読むしかなかったのさ。だから国語の読解力は大切なんだ。今はインターネットで検索すれば、殆どのことを学ぶことができるから幸せだよな。でも、それが正しい記述か、間違った記述か判断する力が必要になってきた。やはり、高校時代の英・数・国・理・社（特に歴史）はしっかり勉強しておいた方がいいかな。総合的な判断力を身につける、という意味でね。オイラは、ダマスのが商売だから、科学的な装いをしてダマスからね。疑り深く読んでね。

タヌキ！頭を休める為に少し休憩するか。

次は第７話