Chapter 2

TCP通信の基礎

Chapter 2では、TCPを利用して通信を行うプログラムを書く方法の概要を示します。

インターネットで利用される通信のほとんどがTCPによって行われています。まず最初にTCPによる通信プログラミング概要を示し、次に単純なサーバとクライアントのサンプルコードを示します。

また、よくある注意点などを解説したうえで、最後は疑似Webサーバとクライアントまでを作成します。

2-1 TCPによるプログラミングの流れ

TCPによる通信は、サーバとクライアントの2者間で行われます。クライアントがサーバに対して接続要求を出すことから始まり、サーバは通信要求が届くまで待ち続けます。サーバが接続要求を受け付けると、クライアントとサーバの間に仮想的な接続（バーチャルサーキット）ができあがります。

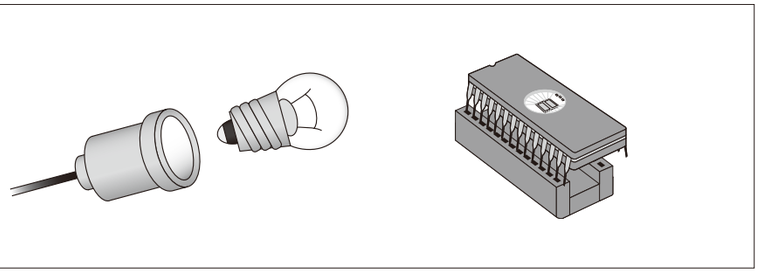
ユーザがバーチャルサーキットに対してデータを送ると、バーチャルサーキットの反対側へデータがそのまま転送されます。そのため、バーチャルサーキットの両側のユーザは、通信路上でのパケットロスなどといった障害への対応を気にすることなく、書き込みと読み出しで通信が可能になっています。

ソケットとプログラミング

サーバとクライアントを結ぶ仮想的な接続を実現するのが「ソケット（socket）」であり、プログラミングでソケットを利用するときに使うのが「ソケットAPI（Application Programming Interface）」です。このソケットAPIはPOSIXという規定で決められており、多くのシステムで同じ記述が可能です[注2-1]。

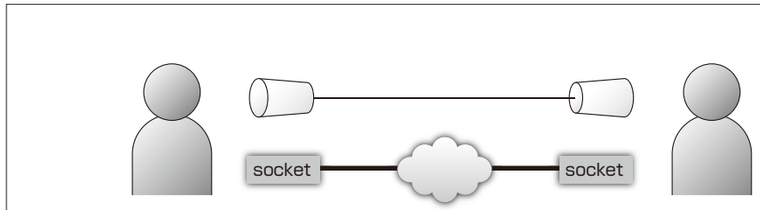
「socket」という単語は電球を接続する端子や、コンセントを表す英単語です。さまざまなものに合わせて接続が可能な「何か」という意味があります。

図2-1　ソケットの概念



ソケットAPIにおいて、ソケットはユーザにとってのデータの出入り口です。また、ソケットは2つ以上のソケットが互いに関係を持つことではじめて有効になります。たとえば、糸電話通信は、両端の紙コップを糸で繋いではじめて利用可能になります。ソケットも同様で、通信相手のソケットと仮想的な関係を持ってはじめて利用可能になります。通信方式によって関係の持ち方や関係を持つ相手の数などが異なるだけです。

図2-2　糸電話のようなソケット



お互い関係を持っている場合、片方のソケットに書き込んだデータはもう片方のソケットから出てきます。ユーザはソケットの裏側で動いている複雑な通信プロトコルなどの仕組みを意識する必要がありません。

ほとんどの通信プログラムは基本的にソケットを使いますが、プログラムによってその実装方法にはいろいろな違いがあります。たとえば、TCP通信を行うプログラムでは、サーバ側とクライアント側で実装手法が異なります。

Linuxにおけるソケットの作成

Linuxでは、socket()システムコールを利用してソケットを作成します（システムコールについては32ページコラム参照）。

ソケットにもさまざまな種類があり、どのようなソケットを作りたいのか、最初に指定しなければなりません。この指定は、socket()システムコールの引数として渡します。

List 2-1　socket()システムコール

```
#include <sys/socket.h>

sockfd = socket(
  int socket_family,    /* アドレスファミリ  */
  int socket_type,      /* ソケットタイプ    */
  int protocol          /* プロトコル        */
);
```

通信路で使われるプロトコルは、「アドレスファミリ」「ソケットタイプ」「プロトコル」の3つ

注2-1：ただし、システムによって多少の違いもあります。

の組み合わせにより決定します。socket()システムコールも、それにあわせて3つの引数を取ります。

ひとつ目の引数(socket_family)がアドレスファミリを表しています。ここでよく指定されるものに、表2-1のものがあります。アドレスファミリとは、ソケットが利用するアドレス体系を示すものです。たとえば、IPv4であればAF_INET、IPv6であればAF_INET6となります。アドレスファミリが異なるソケットは、アドレス体系を含む通信体系がそれぞれまったく別物になります。

表2-1　アドレスファミリ

アドレスファミリ	内容
AF_INET	IPv4によるソケット
AF_INET6	IPv6によるソケット
AF_UNIX	ローカルなプロセス間通信用のソケット。AF_LOCALとも呼ばれる
AF_PACKET	デバイスレベルのパケットインターフェース

2つ目の引数(socket_type)がソケットタイプを表します。ソケットタイプはソケットの性質を表しています。Linuxで利用可能なソケットタイプとしては以下のようなものがあります。

表2-2　ソケットタイプ(man 2 socket より一部抜粋)

ソケットタイプ	解説
SOCK_STREAM	順序性と信頼性があり、双方向の接続されたバイトストリーム(byte stream)を提供する。帯域外(out-of-band)データ転送メカニズムもサポートされる
SOCK_DGRAM	データグラム(接続、信頼性なし、固定最大長メッセージ)をサポートする
SOCK_SEQPACKET	固定最大長のデータグラム転送パスに基づいた順序性、信頼性のある双方向の接続に基づいた通信を提供する。受け取り側ではそれぞれの入力システムコールでパケット全体を読み取ることが要求される
SOCK_RAW	生のネットワークプロトコルへのアクセスを提供する
SOCK_RDM	信頼性はあるが、順序は保証しないデータグラム層を提供する
SOCK_PACKET	廃止されており、新しいプログラムで使用してはいけない

このソケットタイプとソケットファミリの組み合わせによって実際の通信方式が決定されます。たとえば、AF_INETとSOCK_STREAMであればIPv4＋TCPによる通信が行われ、AF_INETとSOCK_DGRAMであればIPv4＋UDPによる通信が行われます。IPv6を利用する場合には、AF_INET6+SOCK_STREAMでIPv6＋TCPの通信が行われ、AF_INET6＋SOCK_DGRAMでIPv6＋UDPの通信が行われます。

AF_UNIX＋SOCK_STREAM、AF_UNIX＋SOCK_DGRAM、AF_INET＋SOCK_RAW(RAWソケット)などもありますが、ここでは割愛します(AF_UNIXはChapter 6で、SOCK_RAWはChapter 12で解説します)。ソケットタイプのうち代表的でもっとも多く利用されるのが「SOCK_STREAM」と「SOCK_DGRAM」の2つです。

SOCK_STREAMは信頼性のある通信を実現します。「信頼性がある」とは、データ送信側で送信したデータが受信側でそのまま届くということです。インターネットそのものは信頼性がなく、途中でパケットが喪失したり、送信したパケットの到着順序がバラバラになる可能性もありますが、SOCK_STREAM型ソケットはカーネル内で喪失したパケットの再送要求や並べ替えを行ってくれます。

一方で、SOCK_DGRAMは信頼性がなく、到着順序も変わる可能性があります。そのため、送信側が送ったつもりでも受信側に届いていないこともあります。順序が変わったり、データが知らないうちに途中で消えるような通信路は使いにくいと思うかもしれません。しかし、音声通話などのように「データが完全に届くこと」よりも「データが即座に届くこと」が優先されるような通信では有効です。

たとえばAF_INET＋SOCK_STREAM型のようにTCPでパケット再送を行うことで信頼性を確保している場合、パケットが喪失を解決するために再送を行ったり、パケットの順序が変わったために並べ替えを行うなどの作業をカーネル内で行うことになり、ユーザプログラムがデータを受け取るまでに時間がかかってしまう可能性があります。SOCK_DGRAM型は「そんなことはいいから早くデータをください」という場合に便利です。

ほかにも、SOCK_STREAMは一度に送信できるデータサイズの制限はありませんが、SOCK_DGRAMは一度に送れるデータの最大長が有限であるという制約もあります(表2-3)。

表2-3　２つのソケットタイプの違い

ソケットタイプ	信頼性	パケットの到着順序	一度に送信できるデータサイズ
SOCK_STREAM	あり	変化なし	制限なし
SOCK_DGRAM	なし	変化する可能性あり	制限あり

socket()システムコールにおける3つ目の引数(protocol)は、プロトコルを表します。利用可能なプロトコルは、ソケットファミリとソケットタイプの組み合わせによっても変わります。この組み合わせが単一のプロトコルのみをサポートする場合は、3つ目の値を指定しなくても自明であるため「0」とすることが可能です。

IPにおいて利用可能なプロトコル番号は、/etc/protocolsに記載されています。

```
$sudo cat ./etc/protocols

# Internet (IP) protocols
#
# Updated from http://www.iana.org/assignments/protocol-numbers and other
# sources.
# New protocols will be added on request if they have been officially
# assigned by IANA and are not historical.
# If you need a huge list of used numbers please install the nmap package.

ip      0       IP              # internet protocol, pseudo protocol number
#hopopt 0       HOPOPT          # IPv6 Hop-by-Hop Option [RFC1883]
icmp    1       ICMP            # internet control message protocol
igmp    2       IGMP            # Internet Group Management
```

```
ggp     3       GGP             # gateway-gateway protocol
ipencap 4       IP-ENCAP        # IP encapsulated in IP (officially ``IP'')
st      5       ST              # ST datagram mode
tcp     6       TCP             # transmission control protocol
egp     8       EGP             # exterior gateway protocol
igp     9       IGP             # any private interior gateway(Cisco)
pup     12      PUP             # PARC universal packet protocol
udp     17      UDP             # user datagram protocol
hmp     20      HMP             # host monitoring protocol
xns-idp 22      XNS-IDP         # Xerox NS IDP
rdp     27      RDP             # "reliable datagram" protocol

.....

(以下略)
```

ソケット作成の実装例

それでは、socket () システムコールを使ったソケット作成サンプルプログラムを作ってみます。ここではAF_INET (IPv4) とSOCK_STREAMという組み合わせでソケットを作成しています。前述のとおり、このAF_INET+SOCK_STREAMという組み合わせは、IPv4によるTCPを表しています[注2-2]。

List 2-2　IPv4 + TCP によるソケット実装例

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>

int
main()
{
  int sock;

  sock = socket(AF_INET, SOCK_STREAM, 0);
  if (sock < 0) {
    printf("socket failed\n");
    return 1;
  }

  return 0;
}
```

このように、すべての通信はsocket () システムコールが返す「ファイルディスクリプタ」を使って行われます。通信だけではなくソケットへの操作などもソケットファイルディスクリプタを利用して行われます。Linuxでは、open () システムコールを利用してファイルを開いたときにできるファイルディスクリプタと同様、ソケットのファイルディスクリプタもintで表現されます。

socket()システムコールは、失敗すると-1を返し、このときのエラー内容はグローバル変数errnoに格納されます(詳しくはChapter 2-3参照)。

ここで注意しなくてはならないのは、ファイルディスクリプタが「0」となっていても、それは正常な値であることです。たとえば、0番でopenされているファイルディスクリプタがない状態でsocket()システムコールを利用した場合、socket()システムコールは0という整数値を返します。そのため、たとえば以下のようなエラー処理を行っているとバグを発生させる可能性があります。

List 2-3　間違ったエラー処理

```
if ((soc = socket(AF_INET, SOCK_DGRAM, 0)) <= 0) {
  perror("socket");
  return -1;
}
```

ここでのポイントは、「<= 0」であるところです。本来ならば「< 0」としなければなりません。

以下のサンプルでは、socket () システムコールは正常終了し、0というファイルディスクリプタを返します。これは、stdin (標準入力) のファイルディスクリプタが0でsocket()システムコールを開始する前にそれを閉じているためです。

List 2-4　ファイルディスクリプタが 0 となる場合

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>

int
main()
{
  int sock;

  printf("fileno(stdin) = %d\n", fileno(stdin));
  close(0);

  /* sock will be zero, and it is not an error! */
  sock = socket(AF_INET, SOCK_DGRAM, 0);
  printf("sock=%d\n", sock);

  return 0;
}
```

POSIXでは、各プロセスが以下のファイルディスクリプタをあらかじめ保持していることを定義しています。

注2-2：なお、このサンプルはソケットを作成しただけであり、サーバでもクライアントでもありません。

表2-4　POSIX におけるファイルディスクリプタ値

整数値	名前	説明
0	stdin	標準入力
1	stdout	標準出力
2	stderr	標準エラー出力

リスト2-4では、0という整数値を持つファイルディスクリプタ（＝標準入力：stdin）をclose ()しています。それを確認できるように、リスト2-4のサンプルプログラムではfileno (stdin) の結果をprintf ()で表示しています。このfileno (stdin) は、標準入力ファイルディスクリプタの整数値を返すので、0という整数値が得られます。すなわち、最初のclose (0) は標準入力をclose ()していることになります。

close (0) を行ったあとのsocket () システムコールの呼び出し結果を見ると、0という整数値になっていることがわかるでしょう。これは「正常に作成できたソケットを表すファイルディスクリプタの整数値が0であった」ということを示しています。このようなとき、socket () システムコールの0という返り値をエラーにしてしまうと、正常終了しているにも関わらずエラー処理に入ってしまいます。

COLUMN

システムコールとは

システムコールとは、オペレーティングシステム (OS) が提供する機能を利用するためのAPI (Application Programming Interface) です。ユーザがカーネルから提供される何らかのサービスを利用する場合、システムコールを利用しなければなりません。言い換えると、ユーザはシステムコールを使わないとカーネルが管理する資源をいっさい使うことができません。代表的なシステムコールとしてはopen ()、read ()、write () などがあります。

一方で、システムコールを内部で利用したライブラリ関数もあります。たとえば、malloc ()やgetaddrinfo () などはシステムコールと勘違いされがちですが、実際にはライブラリ関数です。

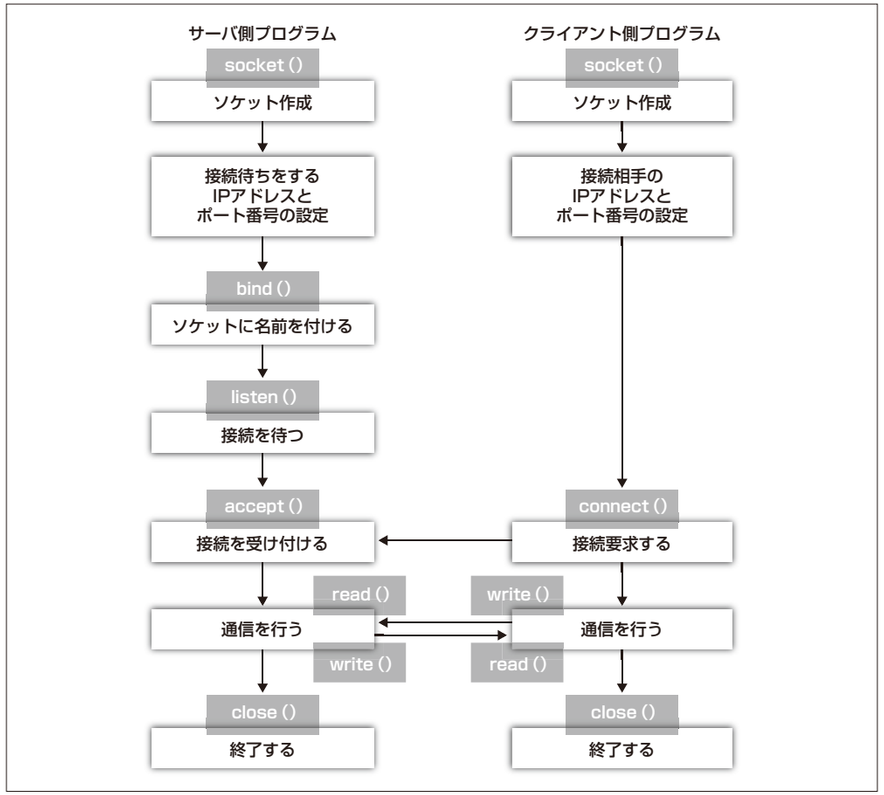
manコマンドで「2」と書いてあるのがシステムコールで、「3」と書いてあるのがライブラリ関数だと覚えておくと、システムコールであるかないかを簡単に確認できます。

2-2 TCPサーバ/クライアントの実装

次はいよいよ、ソケットを利用して実際に通信を行うプログラムを書いてみましょう。Chapter 2では、これ以降TCP通信について解説していきます。

TCPによる通信を行うとき、サーバとクライアントという役割分担があります。サーバは特定のポートでクライアントからのコネクション要求を待ち、クライアントはサーバが待っているポートに接続要求を出します。サーバが接続要求を受け付けられるようにするには、bind ()、listen ()、accept () の3つのシステムコールを利用します。

図2-3　TCP 通信のプログラミング



bind () はソケットに名前を付けることによって待ち受けを行うポート番号を明示するために利用されます。

次に行われるlisten () によって、サーバ側は待ち受け状態へと入ります。待ち受け状態へと入ったサーバに対して、クライアントがconnect () システムコールで接続要求を出します。

サーバ側でクライアントからのTCP接続要求を受け付けると、ブロックしていたaccept () システムコールが返り、新しいソケットがサーバ側で作成されます。このソケットは、クライアントとのTCP接続が成功したことを表します。そして、サーバ側でもう一度accept () システム

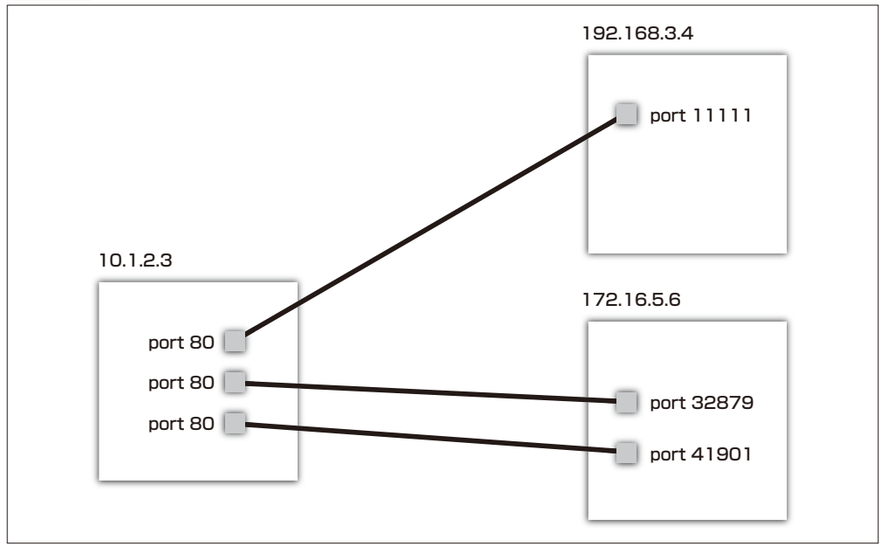
コールを利用すると、次のクライアントからのTCP接続を待てます。

このように、ひとつのサーバは複数のクライアントからのTCP接続を受け付けることができます。TCPによる接続は、

- 相手のIPアドレス
- 自分のIPアドレス
- TCP宛先ポート番号
- TCP送信元ポート番号

を利用して一意性が保たれます。TCPにポート番号の組があるのは、同一の機器同士が複数のTCP接続を同時に張れるようにするためです。

図2-4 TCP接続の一意性



単純な TCP サーバの実装

最初に、単純なTCPサーバを実装します。このTCPサーバは、接続してきたクライアントに対して「HELLO」という文字列を送信して終了します。

サーバプログラミングの手順

TCP通信を行うサーバプログラムを書くには、以下のような手順を踏む必要があります。

- ソケットを作る
- 接続待ちをするIPアドレスとポートを設定する
- ソケットに名前を付ける（bind()する）
- 接続を待つ
- クライアントからの接続を受け付ける
- 通信を行う

このように、サーバはクライアントからの接続要求を待ちます。このとき、接続待ちをするTCPポート番号など、「どのような待ち方をするか」を設定しないといけません。一度接続ができあがってしまえば、ソケットの利用方法はサーバとクライアントで通信方法に違いはありません。どちらからも同様にデータを送信/受信できます。また、どちらか一方がclose () システムコールを利用すれば通信が終了するため、その処理もまったく同じです。

Linuxにおける単純なTCPサーバのサンプルコードがList 2-5です。このTCPサーバの使い方はクライアントと一緒にのちほど説明します。なお、コードを簡単にするため、エラー処理は省いてあります。

List 2-5 単純な TCP サーバの実装

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int
main()
{
  int sock0;
  struct sockaddr_in addr;
  struct sockaddr_in client;
  int len;
  int sock;

  /* ソケットの作成 */
  sock0 = socket(AF_INET, SOCK_STREAM, 0);

  /* ソケットの設定 */
  addr.sin_family = AF_INET;
  addr.sin_port = htons(12345);
  addr.sin_addr.s_addr = INADDR_ANY;
  bind(sock0, (struct sockaddr *)&addr, sizeof(addr)); ①

  /* TCPクライアントからの接続要求を待てる状態にする */
```

```
  listen(sock0, 5);

  /* TCPクライアントからの接続要求を受け付ける */
  len = sizeof(client);
  sock = accept(sock0, (struct sockaddr *)&client, &len); ❷

  /* 5文字送信 */
  write(sock, "HELLO", 5); ❸

  /* TCPセッションの終了 */
  close(sock);

  /* listen するsocketの終了 */
  close(sock0);

  return 0;
}
```

まずはソケットを作ってIPアドレスとポートを設定、名前を付けます(bind()します)(❶)。そのあと、接続を待ってクライアントからの接続を受け付け(❷)、送信して終了(❸)、という流れになっています。TCPセッションのソケットとTCPコネクションを待ち受けるソケットを、それぞれclose()しているところに注意してください。

単純なTCPクライアントの実装

次は、この単純なTCPサーバと接続する単純なTCPクライアントを実装します。

クライアントプログラミングの手順

クライアント側で行うプログラミングの手続きは以下のとおりです。

- ■ソケットを作る
- ■接続相手を設定する
- ■接続する
- ■通信を行う

クライアント側は、特定のIPアドレスとTCPポート番号(接続待ちをしているサーバ)に対して「接続要求」を出します。サーバから返信を受け取り、接続が成功すると通信を開始できます。一度接続ができあがってしまえば、サーバとクライアントで通信方法に違いはありません。

単純なTCPクライアントのサンプルコードをList 2-6に示します。このTCPクライアントは、サーバに接続すると文字列を受信して表示します。

List 2-6 単純なTCPクライアントの実装

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int
main()
{
  struct sockaddr_in server;
  int sock;
  char buf[32];
  int n;

  /* ソケットの作成 */
  sock = socket(AF_INET, SOCK_STREAM, 0);

  /* 接続先指定用構造体の準備 */
  server.sin_family = AF_INET;
  server.sin_port = htons(12345);
  /* 127.0.0.1はlocalhost */
  inet_pton(AF_INET, "127.0.0.1", &server.sin_addr.s_addr);

  /* サーバに接続 */
  connect(sock, (struct sockaddr *)&server, sizeof(server));

  /* サーバからデータを受信 */
  memset(buf, 0, sizeof(buf));
  n = read(sock, buf, sizeof(buf));

  printf("%d, %s\n", n, buf);

  /* socketの終了 */
  close(sock);

  return 0;
}
```

サーバ側のサンプルプログラム同様、コードを簡単にするためにエラー処理は省いてあります。

これらの利用方法ですが、まず「単純なTCPサーバ」側のプログラムを実行してください。サーバ側のプログラムが実行された状態で「単純なTCPクライアント」側のプログラムを実行すると「HELLO」という文字列のやり取りが行われます。

TCPクライアントのコード中にある「127.0.0.1」は「localhost(自分自身)」を表しています。そのため、サーバとクライアント両方のプログラムを同一ホスト上で実行しなければなりません。クライアントプログラムでサーバのIPアドレスを指定している部分を適切な値に変更すれば、別ホストでの通信が可能になります(注2-3)。

注2-3：自分のIPアドレスを知りたい場合には「ifconfig -a」コマンドを利用します。

通信の終了

TCPによる通信を終了するにはclose () システムコールを利用します。このとき、close () を行ったソケットの通信相手側でのread () システムコールは、「EOF (End Of File：ファイルの最後まで読み込んだ)」を意味する「0」という値を返します。

よって、read () システムコールが「0」という値を返したときにはTCP接続が相手からclose () によって切断されたと想定してプログラムを作成する必要があります。相手側でclose () が呼ばれたのではなく、何らかの原因によって通信に対して障害が発生した場合にはread () から「-1」が返り、障害内容はerrnoに記述されます (errnoに関しては後述)。

なお、read () システムコールの第三引数に「0」という値を入れてしまうと、返り値も「0」となるので注意が必要です。何らかのバグでread () の第3引数に渡す変数の中身が0になってしまった結果、read () が0という値を返したことで正常終了パスへとプログラムが入ってしまうバグが発生し、デバッグ時に「何でここで通信が切れてるのだろう？」と見当違いの原因究明をしてしまう場合があります。

COLUMN

サンプルプログラムの実行について

このサンプルを実行するためには、サーバとクライアント両方のソースコードをコンパイルして実行するわけですが、同じディレクトリに両方のソースコードを入れて単純にgccでコンパイルすると、両方とも「a.out」という実行ファイル名になってしまい、先に生成した実行ファイルが上書きされてしまいます。これでは両方同時に実行できないため、サーバとクライアント両方を同じディレクトリ上でコンパイルしたい場合には、a.out以外のファイル名でコンパイル結果を出力してください。

たとえば、サーバ側プログラムを「server」という名前の実行ファイルにしたい場合には、

```
gcc -o server serversample.c
```

のようにコンパイルを実行します。同様に、クライアント側プログラムを「client」という名前の実行ファイルにしたい場合には、

```
gcc -o client clientsample.c
```

のようにコンパイルしてください。

同じホスト内での通信だけでは「通信を行っている」という実感がわきにくいと思います。ぜひ別々のホストでサーバとクライアントを実行して試してみてください。

2-3 ソケットプログラミングのエラー処理

さて、最初の通信プログラムはうまく動いたでしょうか？　ソケットプログラミングにおいてエラーが発生したとき、その原因を知ることはデバッグなどの観点から非常に重要です。ここでは、エラー内容の取得方法を説明します。

errno と perror ()

システムコールのエラー内容は直接返り値に反映されるわけではなく、変数「errno」に格納されます。そのため、システムコールのエラー (基本的に「-1」) を確認したら、次にerrnoを参照することで、エラー処理を行っていきます (List 2-7)。

List 2-7　errno によるエラー処理

```
#include <errno.h>

…

if (socket() < 0) {
  if (errno == … ) { … }
}
```

システムコールがどのようなエラーを発生させるのか (＝errnoにどのような値があるのか) は、manコマンドで調べます。たとえば、socket () システムコールで発生し得るエラーを知るには、

```
% man 2 socket
```

のように実行します。

以下は、代表的なエラーとerrnoです。

表2-5　socket () システムコールで発生する代表的なエラーと errno (man ファイルより抜粋)

errno	エラー内容
EACCES	指定されたタイプまたはプロトコルのソケットを作成する許可が与えられていない
EAFNOSUPPORT	指定されたアドレスファミリーがサポートされていない
EINVAL	知らないプロトコル、または利用できないプロトコルファミリである

表2-5　socket()システムコールで発生する代表的なエラーとerrno（manファイルより抜粋）（続き）

errno	エラー内容
EMFILE	プロセスのファイルテーブルが溢れている
ENFILE	カーネルに新しいソケット構造体に割り当てるための十分なメモリがない
ENOBUFS または ENOMEM	十分なメモリがない。十分な資源が解放されるまではソケットを作成できない
EPROTONOSUPPORT	このドメインでは指定されたプロトコルまたはプロトコルタイプがサポートされていない

ソケット作成の失敗とperror()利用例

errnoの値だけではわかりにくく、エラー内容を文字列で表示したいこともあります。このようなときは、perror()という関数を利用すると、エラー内容を標準エラー出力に書き出してくれます。

List 2-8　perror()関数

```
#include <stdio.h>

void perror(
  const char *string     /* 前置きメッセージ */
);
```

では、実際にソケット作成が失敗するのはどのようなときでしょうか。socket()システムコールを失敗させたあとにperror()を使ってみましょう。

List 2-9　socket()システムコールを失敗させる

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>

#include <errno.h>

int
main()
{
  int sock;

  sock = socket(3000, 4000, 5000);
  if (sock < 0) {
    perror("socket");
    printf("%d\n", errno);
    return 1;
  }
  return 0;
}
```

ここでは、変な値を渡したためにsocket()システムコールが失敗しています。失敗するとファイルディスクリプタに-1が返り、if文の中に入ります。

そこでは、まずperror()が呼ばれ、

```
socket: Socket type not supported
```

と表示されます。エラーメッセージ中の「:」より前の部分は、perror()に渡す引数により変わります。たとえば、

```
perror("hogehoge");
```

とすると、

```
hogehoge: Socket type not supported
```

のように表示されます。

このサンプルでは、続いてprintf()を使ってerrnoの値も表示しています。表示される値はerrno.hにおいて「ESOCKNOSUPPORT」としてdefineされている値になります(注2-4)。

プログラムを書くときにはエラー処理は非常に重要です。perror()やerrnoを活用してデバッグや運用・管理のしやすいプログラミングを心がけてほしいと思います。

perror()利用上の注意点

エラー処理で注意しなければならないのが、errnoやperror()が反映している値は「最後のエラー内容」である点です。

たとえば以下のようなプログラムがあるとします。プログラマが❶側のエラーを得たいと思っていた場合、プログラマの意図とは違った結果が返ります。

List 2-10　注意すべきエラー処理

```
#include <stdio.h>
#include <sys/socket.h>
```

注2-4：厳密にはerrno.hからincludeされるファイルに書いてあるESOCKNOSUPPORTかもしれません。

```
#include <errno.h>
#include <unistd.h>

int
main()
{
  int sock;
  sock = socket(AF_INET, 4000, 5000); ❶

  write(-1, "hoge", 4); ❷
  if (sock < 0) {
    perror("socket");
    return 1;
  }

  return 0;
}
```

これを実行すると、

```
socket: Bad file descriptor
```

となります。

上記サンプルプログラムでは、❶のsocket () システムコールのエラーは「EAFNOSUPPORT (指定されたアドレスファミリがサポートされていない)」になります。一方で、❷のwrite () システムコールのエラーは「EBADF (不正なファイルディスクリプタ)」です。

上記を実行してわかるように、perror()とprintf()による結果は、最後に行われた❷の方が表示されます。一見当たり前のようですが、このような間違いがバグとして混入すると、なかなか発見できない場合があるので気を付けましょう。

printf()と perror()の実行順

結論から先にいうと、printf()をperror()の前に実行してはいけません。

List 2-11は先ほどのサンプルプログラムと本質は同じですが、もう少し複雑なケースです。printf () 関数のなかでほかのシステムコールが利用されており、そのシステムコールがエラー終了してerrnoを上書きしてしまうというものです。このような間違いはよくあります。

List 2-11　printf を perror の前に実行したケース

```
#include <stdio.h>
#include <sys/types.h>
```



```
#include <sys/socket.h>

#include <errno.h>

int
main()
{
  int sock;

  sock = socket(3000, 4000, 5000);
  if (sock < 0) {
    close(fileno(stdout)); ❶
    printf("%d\n", errno);

    perror("socket");
    return 1;
  }

  return 0;
}
```

ここではprintf()関数が失敗する状態を作りつつ、perror()の前にprintf()を行っています。

❶では、標準出力へのファイルディスクリプタをclose()しています。そのため、printfを呼び出すとprintf()内で標準出力に書き込もうとするシステムコールが「EBADF」によって失敗し、errnoはprintf()内部での失敗内容を反映(上書き)してしまいます。その後、perror()が呼び出されるとEBADFがerrnoにセットされたものとしてエラー内容が表示されます。

上記例はprintf()ですが、printf()以外の関数であっても同様の問題が発生することがあります。perror()やerrnoの利用には細心の注意が必要です。

bind()の意味

ここまでのサンプルプログラムでは、サーバ側でbind()を行ってきました。このbind () については「名前を付ける」という説明を行ってきましたが、これだけでは意味がわかりにくいので、あえて「bind()を使わないとどうなるか」という事例をここで紹介します。

List 2-12のサンプルプログラムは、bind()を利用せずにlisten()を行っています。このサンプルプログラムは、待ち受けポート番号をprintf()で表示します。たとえば、以下のような結果が実行時に表示されます。

```
% ./a.out
0.0.0.0 : 56664
```

「56664」とあるのがサーバの待ち受けポート番号ですが、これは実行するたびに変わります。このTCP 56664番ポートにTCPコネクションを確立すると、サーバは接続相手に対してwrite()によって「HOGE\n」という5文字を送信します。

クライアントが、このサンプルプログラムと接続するようすを簡単に試すには、telnetコマンドが便利です。たとえば、ポート番号が56664であるとき、以下のようになります。

```
% telnet localhost 56664
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
HOGE
Connection closed by foreign host.
```

localhostに接続直後にサンプルプログラムから「HOGE\n」という5文字を受け取り、TCPコネクションがサンプルプログラム側から切断されているのがわかります。

List 2-12　bind ()を行わない場合

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

/* ポート番号とbindされたアドレスを表示する関数 */
void
print_my_port_num(int sock)
{
  char buf[48];
  struct sockaddr_in s;
  socklen_t sz;

  sz = sizeof(s);

  /* ソケットの「名前」を取得、getsockname()はChapter6参照 */
  if (getsockname(sock, (struct sockaddr *)&s, &sz) != 0) {
    perror("getsockname");
    return;
  }

  /* bindされているIPアドレスを文字列へ変換 */
  inet_ntop(AF_INET, &s.sin_addr, buf, sizeof(buf));

  /* 結果を表示 */
  printf("%s : %d\n", buf, ntohs(s.sin_port));
}

int
```



```
main()
{
  int s0, sock;
  struct sockaddr_in peer;
  socklen_t peerlen;
  int n;
  char buf[1024];

  /* ソケットを作成していきなりlisten()する */
  s0 = socket(AF_INET, SOCK_STREAM, 0);

  if (listen(s0, 5) != 0) {
    perror("listen");
    return 1;
  }

  /* listen()すると自動的に未使用ポートを割り当てられることを確認 */
  print_my_port_num(s0);

  /* TCPコネクションを受付 */
  peerlen = sizeof(peer);
  sock = accept(s0, (struct sockaddr *)&peer, &peerlen);
  if (sock < 0) {
    perror("accept");
    return 1;
  }

  /* 相手に文字列を送信して終了 */
  write(sock, "HOGE\n", 5);

  close(sock);
  close(s0);

  return 0;
}
```

bind()を利用せずに自動的にポート番号割り当てを行うケースとして、たとえばP2Pなどが挙げられます。あえてbind()を行わないことによって、システム内の利用されていない待ち受けポートがカーネルによって選択されます。

また、意識しないことが多いと思いますが、connect()を行うクライアント側でもbind()を利用せずにTCPコネクションを確立しています。connect()が呼ばれると、自動的にソケットに対応するポート番号割り当てが行われます。本書のサンプルプログラムでは利用していませんが、逆に、bind()を行ってローカル側のポート番号を明示的に設定しつつ、connect()を行うことも可能です。

このように、bind()を行なわなくても自動的にポート番号などが割り当てられますが、サーバがプログラマの意図する特定のポート番号で待っていることも重要です。たとえば、Webでは、とくに明示的にポート番号を指定しなければ「サーバ側がTCPの80番で待っている」こと

を前提に通信を行います。

このサンプルプログラムによって、bind()を行うことによって明示的にソケットに「名前を付ける」という処理の意味を理解できるでしょう。

listen()の意味

次はlisten()の意味について解説します。

TCPのセッションを表現しているソケットを生成するのはaccept()システムコールですが、カーネルがクライアントからのTCPセッションを受け付けるようになるのはlisten()システムコールの利用後になります。

たとえば、一度に3つのTCPコネクション要求が別々のクライアントから到着し、accept()が間に合わない場合があります。このようなとき、カーネルはTCPセッションの準備をあらかじめ行っておき、ユーザアプリケーションがaccept()を行った時点でソケットをユーザアプリケーションに渡しています。

listen()システムコールは以下のように宣言されています。

List 2-13　listen()システムコール

```
#include <sys/types.h>
#include <sys/socket.h>

int listen(
  int sockfd,    /* SOCK_STREAM型のソケット */
  int backlog    /* 接続保留状態を保持できる数 */
);
```

listen()システムコールの第二引数であるbacklogが、accept()されていないTCPコネクションを保持できる最大数になります。この引数からもわかるように、listen()の開始によってクライアントからのTCPコネクションが受け付け可能になります。

accept()は、カーネル内に保持された確立済みTCPセッションを、ソケットという形でユーザアプリケーションに渡すことが主目的であり、accept()そのものがTCPセッション確立を行っているわけではありません。

listen()システムコールは成功時に0を、失敗時に-1を返します。このとき、エラー内容はerrnoに設定されます。errnoの値としは以下の内容が設定される可能性があります。

表2-6　listen()で発生するerrno（man 2 listen より）

errno	内容
EADDRINUSE	別のソケットがすでに同じポートをlisten()している
EBADF	引数sockfdが有効なディスクリプタではない
ENOTSOCK	引数sockfdがソケットではない
EOPNOTSUPP	ソケットはlisten()がサポートしている型ではない

listen()の2番目の引数は、確立されていない不完全なTCPセッション数ではなく、確立されたTCPセッション数を表しているのでご注意ください。ちなみに、古い設計ではlisten()の第二引数は不完全なTCPセッション数を表現していました。しかし、SYN floodingという偽TCP接続要求パケットの大量送信によるサービス不能攻撃が多発したことなどが要因で変更されました。確立されていないTCPセッション数は、sysctlのtcp_max_syn_backlogを参考にしてください。たとえば、以下のコマンドを実行するとIPv4 TCPのtcp_max_syn_backlogを知ることができます。

```
% sysctl net.ipv4.tcp_max_syn_backlog
```

なお、net.ipv4.tcp_syncookiesを有効にすると、tcp_max_syn_backlogの値は利用されなくなり、論理的な上限はなくなります。

無効になったソケットに対するデータ送信

何らかの理由で無効になってしまった(注2-5)ソケットに対して、write()やsend()などのデータ送信用システムコールを実行するとSIGPIPEシグナルが発生します。

シグナルとは、UNIX系OSに含まれる非同期イベント発生を伝えるためのソフトウェア割り込み機構です。普通、シグナルを受け取ったプロセスは、実行を終了して消滅します。

しかし、シグナルを受け取るとプロセスが必ず終了するわけではありません。シグナルを受け取ったときの挙動をあらかじめ規定することで、突然のプロセス終了を防げます。

それには、以下の2つの方法があります。

- SIGPIPE用のシグナルハンドラを指定する
- SIGPIPEを無視するように指定する

SIGPIPE用のシグナルハンドラを指定する

まずは、signal()システムコールを利用してSIGPIPE用のシグナルハンドラを指定する手法です。シグナルハンドラを利用したサンプルプログラムには以下のような部分が含まれます。

List 2-14　シグナルハンドラを利用する

```
#include <signal.h>

void sigfunc(int n)
```

注2-5：相手側がclose()を行ったとき、相手側の読み込みがshutdown()によって閉じられたとき、ネットワーク障害によってTCP接続が破壊されたときなどです。

```
{
  write(fileno(stderr), "hoge", 4);
}

int
main()
{
  ...

  signal(SIGPIPE, sigfunc);

  ...
}
```

上記サンプルプログラムの自作シグナルハンドラであるsigfunc()は、SIGPIPEが発生したときにコールバックされます。このときsigfunc(int n)の変数nには、シグナルの番号が入ります。signal()システムコールで複数のシグナル用のシグナルハンドラとして設定していない場合には、nにはSIGPIPEしか入りません。

このサンプルのプログラムのシグナルハンドラ内では、標準エラー出力に「hoge」と書いてシグナルハンドラは終了しています。シグナルによる割り込みが終了後は、write()などのデータ送信用システムコールは「-1」を返します。そのとき、errnoにはEPIPEが設定されます。

なお、シグナルハンドラの中で利用可能な関数はかぎられています。たとえば、printf()やmalloc()などはシグナルハンドラ内では使ってはいけない関数なのでご注意ください(本Chapter最後のCOLUMNを参照)。

SIGPIPEを無視するように指定する

あるいは、シグナルを無視する設定も可能です。

プロセスがシグナルを無視するようにするには、sigignore()関数を利用します。

List 2-15 sigignore()関数

```
#include <signal.h>

sigignore(
  int SIGPIPE
);
```

sigignore()関数を利用してSIGPIPEシグナルを無視するように設定するには、以下のようにします。

List 2-16 SIGPIPE シグナルを無視する

```
#include <signal.h>

int
main()
{
  ...

  sigignore(SIGPIPE);

  ...
}
```

文字列でのエラー内容取得

perror()関数は自動的に標準エラー出力にエラー内容を記述しますが、標準エラー出力への出力ではなく、文字列としてエラー内容を取得したい場合にはstrerror()関数が利用できます。

List 2-17 strerror()関数

```
#include <string.h>

char *strerror(
  int errnum        /* エラー番号 */
);
```

引数errnumは、説明文字列を得たいエラー番号です。

しかし、strerror()はperror()と同様にスレッドセーフではありません。スレッドセーフにエラー番号の説明文字列を得るには、strerror_r()関数を利用します。

List 2-18 strerror_r()関数

```
#include <string.h>

int strerror_r(
  int errnum,       /* エラー番号 */
  char *strerrbuf,  /* 文字列格納用バッファ */
  size_t buflen     /* strerrbufのサイズ */
);
```

引数errnumは説明文章を得たいエラー番号、strerrbufはエラー説明文字列を格納するバッファ、buflenはstrerrbufのサイズです。strerrbufに格納される文字列は必ずNUL(\0)終端されます。

strerror_r () 関数は、成功時に0を返します。エラー発生時の返り値としては、errnumが知らない値である場合strerrbufに「Unknown error: 」という文字列と番号を記述し、EINVALを返します。buflenが不正な値の場合は、ERANGEを返しつつstrerrbufには何も記述されません。

COLUMN

manと章番号

「man 2 socket」コマンドの2というのはシステムコールを示しています。ライブラリ関数の場合は3になります。これらの数値はman（マニュアル）の章番号です。

manコマンドにおける章番号と内容の対応は以下のようになっています（日本語版 man manより）。

- ■1：実行プログラムまたはシェルのコマンド
- ■2：システムコール（カーネルが提供する関数）
- ■3：ライブラリコール（システムライブラリに含まれる関数）
- ■4：スペシャルファイル（通常 /dev に置かれている）
- ■5：ファイルのフォーマットとその約束事。たとえば /etc/passwd など
- ■6：ゲーム
- ■7：マクロのパッケージとその約束事。たとえばman (7)、groff (7) など
- ■8：システム管理用のコマンド（通常はroot専用）
- ■9：カーネルルーチン［非標準］

「man socket」のように数値部分は指定なしでも説明文が表示されます。「man 7 socket」はLinuxソケットインターフェース全般に関して解説しています。興味がある方はそちらもぜひご覧ください。

2-4 名前解決の実装

インターネットに接続された機器はIPアドレスと呼ばれる数値によって通信を行っていますが、それでは人間がわかりにくいため、一般的には「www.example.com」のような「名前」が利用されます。この名前からIPアドレスへの変換作業、すなわち「名前解決」が通信プログラムを書くときにも重要になります。

昔は、名前解決のために「gethostbyname ()」という関数を利用するのが一般的でした。そのため、多くのプログラミング参考書ではinet_addr () 関数やgethostbyname () 関数を利用した通信プログラムを解説しています。しかし、gethostbyname () 関数はIPv4の名前解決しか行えず、IPv6は扱えないという問題点があります。今後を考えるとIPv4にしか対応していないプログラムを書くべきではありません。

さらに、多くの処理系ではすでにgethostbyname () 関数の代わりにgetaddrinfo () 関数を利用することが推奨されています。Linuxも例外ではありません。たとえば、manの「gethostbyname (3)」にある説明文の最初には、以下のように書かれています。

> **これらの関数は過去のものである。アプリケーションでは、代わりにgetaddrinfo (3) とgetnameinfo (3) を使用すること。**

これらを踏まえ、本書ではIPv6も利用可能なgetaddrinfo () 関数を利用した解説を行います。gethostbyname () やinet_addr () については巻末のAppendixにまとめましたので、必要な方はご覧ください。

名前解決のサンプルプログラム

まず最初に、名前解決を行う単純なサンプルプログラムを示します。

ここでは、話を単純化するためにIPv4のみを対象とします。しかも文字列から32ビットのIPv4アドレス値を取得するという、gethostbyname () 関数と同じような使い方をしています。

List 2-19　単純な名前解決プログラム

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int
main()
{
  char *hostname = "localhost";
  struct addrinfo hints, *res;
  struct in_addr addr;
  int err;

  memset(&hints, 0, sizeof(hints));
  hints.ai_socktype = SOCK_STREAM;
  hints.ai_family = AF_INET;

  if ((err = getaddrinfo(hostname, NULL, &hints, &res)) != 0) {
    printf("error %d\n", err);
    return 1;
  }
```

```
  addr.s_addr = ((struct sockaddr_in *)(res->ai_addr))->sin_addr.s_addr;

  printf("ip address : %s\n", inet_ntoa(addr));

  freeaddrinfo(res);

  return 0;
}
```

getaddrinfo()関数の結果は毎回新しいメモリを確保することで作成されており、getaddrinfo()関数はスレッドセーフに作られています。そのため、getaddrinfo()関数で確保したメモリは不必要になった時点で解放する必要があります。

getaddrinfo()関数によって確保されたaddrinfo構造体は、freeaddrinfo()関数を使って解放します。このfreeaddrinfo()関数を忘れないよう、気を付けましょう。

エラー解析関数

getaddrinfo()関数には特別なエラー解析関数「gai_strerror()」があります。getaddrinfo()関数がエラーで終了したときに、gai_strerror()関数を利用してエラー内容を表示させる単純なサンプルを示します(List 2-20)。

List 2-20　gai_strerror 関数を使ったプログラム

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int
main()
{
  int err;

  if ((err = getaddrinfo(NULL, NULL, NULL, NULL)) != 0) {
    printf("error %d : %s\n", err, gai_strerror(err));
    return 1;
  }

  return 0;
}
```

このサンプルプログラムでは、無効な引数でgetaddrinfo()関数を利用し、変数errが0ではない値になるようにしています。getaddrinfo()関数が失敗したあとには、if文の中でgai_strerror

関数が返すエラー説明文字列をprintf()関数で出力することによりエラー内容を表示しています。著者の環境では、

```
error -2 : Name or service not known
```

という実行結果が表示されました。

IPv6 と IPv4 両方に対応する

次に、名前から得られるIPアドレスをIPv4あるいはIPv6にかぎらず、すべて取得する方法を示します。ソケットファミリに「PF_UNSPEC」を指定するのがポイントです。

List 2-21　IP アドレスをすべて取得する

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int
main()
{
  char *hostname = "localhost";
  char *service = "http";
  struct addrinfo hints, *res0, *res;
  int err;
  int sock;

  memset(&hints, 0, sizeof(hints));
  hints.ai_socktype = SOCK_STREAM;
  hints.ai_family = PF_UNSPEC;

  if ((err = getaddrinfo(hostname, service, &hints, &res0)) != 0) {
    printf("error %d\n", err);
    return 1;
  }

  for (res=res0; res!=NULL; res=res->ai_next) {
    sock = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
    if (sock < 0) {
      continue;
    }

    if (connect(sock, res->ai_addr, res->ai_addrlen) != 0) {
```

```
      close(sock);
      continue;
    }

    break;
  }

  freeaddrinfo(res0);

  if (res == NULL) {
    /* 有効な接続ができなかった */
    printf("failed\n");

    return 1;
  }

  /* ここ以降にsockを使った通信を行うプログラムを書いてください */

  …

  return 0;
}
```

上記サンプルプログラムではconnect()処理まで行っています。getaddrinfo()関数によって得られた結果に応じて順次接続していき、接続が成功したら通信を行うコードへと移行しています。このような書き方をすることで、IPv4かIPv6のどちらで通信しているかをまったく気にせずに通信プログラムを記述できます。

実際にIPv4とIPv6のどちらで通信が行われるのかは、手元の環境設定やサーバ、DNSの設定によって変わります。

getaddrinfo()をbind()で使う

AI_PASSIVEフラグを指定してgetaddrinfo()を利用することで、bind()のためのsockaddr構造体を作成することもできます。

getaddrinfo()のAI_PASSIVEフラグを利用する利点としては、INADDR_ANYとin6addr_anyを切り分けたり、sockaddr_in構造体とsockaddr_in6構造体を個別に考えなくてもよいという点が挙げられます。

AI_PASSIVEでgetaddrinfo()を利用するときには、getaddrinfo()の第一引数はNULLで、第二引数にポート番号を渡します。そのとき、getaddrinfo()の第二引数は整数ではなく文字列なのでご注意ください。

以下に、先に示した単純なTCPサーバ(List 2-4)をgetaddrinfo()化したサンプルを示します。基本的な流れはList 2-4と変わりません。

List 2-22 単純なTCPサーバ(getaddrinfo()版)

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

int
main()
{
  int sock0;
  struct sockaddr_in client;
  socklen_t len;
  int sock;
  struct addrinfo hints, *res;
  int err;

  memset(&hints, 0, sizeof(hints));
  hints.ai_family = AF_INET;
  hints.ai_flags = AI_PASSIVE;
  hints.ai_socktype = SOCK_STREAM;
  err = getaddrinfo(NULL, "12345", &hints, &res);
  if (err != 0) {
    printf("getaddrinfo : %s\n", gai_strerror(err));
    return 1;
  }

  /* ソケットの作成 */
  sock0 = socket(res->ai_family, res->ai_socktype, 0);
  if (sock0 < 0) {
    perror("socket");
    return 1;
  }

  if (bind(sock0, res->ai_addr, res->ai_addrlen) != 0) {
    perror("bind");
    return 1;
  }

  freeaddrinfo(res); /* addrinfo構造体を解放 */

  /* TCPクライアントからの接続要求を待てる状態にする */
  listen(sock0, 5);

  /* TCPクライアントからの接続要求を受け付ける */
  len = sizeof(client);
  sock = accept(sock0, (struct sockaddr *)&client, &len);

  /* 5文字送信 */
  write(sock, "HELLO", 5);
```

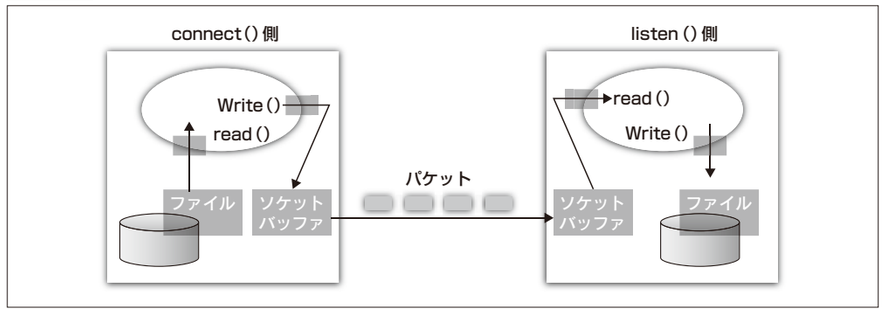
```
  /* TCPセッションの終了 */
  close(sock);

  /* listen するsocketの終了 */
  close(sock0);

  return 0;
}
```

2-5 単純なファイル転送プログラム

次は、TCPによる通信そのものに関する理解を深めるために、単純なファイル転送プログラムを紹介します。

このサンプルプログラムでは、connect () を行う側がファイルを送信し、listen () を行う側がファイルを受け取ります。図2-5に、ファイル転送プログラムの動作を示します。

図2-5　TCP 通信のプログラミング



まず、ファイル送信側はファイルからデータを読み込むためにopen () を行います。ネットワークを通じたファイル送信用にはソケットが用意されます。その後、ファイル読み込み用のファイルディスクリプタからデータをread () しつつ、その結果をソケットに対してwrite () していきます。

このとき、write () されたデータは直接ネットワークへと送信されるわけではなく、カーネル内のソケットバッファと呼ばれるバッファへとコピーされます。ソケットバッファへ格納されたデータは、ネットワークの形態に合わせたサイズへと小分けにされ、パケットとして送信されていきます。TCPには、途中ネットワークで喪失したパケットを検知して再送信する仕組みや、ネットワーク上の混雑を回避する輻輳制御機構がありますが、それらの仕組みが動作しながらパケット化されたソケットバッファ内のデータが送信されていきます。

listen () を行っているファイル受信側は、最初に保存用のファイルを作成します。次に、ネットワークを通じたファイル受信用にソケットが用意されます。

ファイル送信側からのconnect () が行われ、ファイル受信側でlisten () しているソケットからaccept () が完了したあとに、ファイル受信側はファイル送信側からのファイルデータをread () しつつ、その結果をファイルへとwrite () します。

ファイル送信側からのパケットは、パケットとして直接read () されるわけではなく、一度ソケットバッファに格納されてからread () される点にご注意ください。

ファイル送信側サンプルプログラム

次は、実際のサンプルプログラムです。まずは、connect () を行っているファイル送信側です。

List 2-23　ファイル送信側

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <fcntl.h>

int
main(int argc, char *argv[])
{
  char *service = "12345";
  struct addrinfo hints, *res0, *res;
  int err;
  int sock;
  int fd;
  char buf[65536];
  int n, ret;

  if (argc != 3) {
    fprintf(stderr, "Usage : %s hostname filename\n", argv[0]); ❶
    return 1;
  }

  fd = open(argv[2], O_RDONLY); ❷
  if (fd < 0) {
    perror("open");
    return 1;
  }
```

```
memset(&hints, 0, sizeof(hints));
hints.ai_socktype = SOCK_STREAM;
hints.ai_family = PF_UNSPEC;
if ((err = getaddrinfo(argv[1], service, &hints, &res0)) != 0) {
  printf("error %d : %s\n", err, gai_strerror(err));
  return 1;
}

for (res=res0; res!=NULL; res=res->ai_next) {
  sock = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
  if (sock < 0) {
    continue;
  }

  if (connect(sock, res->ai_addr, res->ai_addrlen) != 0) {
    close(sock);
    continue;
  }

  break;
}

freeaddrinfo(res0);

if (res == NULL) {
  /* 有効な接続ができなかった */
  printf("failed\n");
  return 1;
}

while ((n = read(fd, buf, sizeof(buf))) > 0) {
  ret = write(sock, buf, n);
  if (ret < 1) {
    perror("write");
    break;
  }
}

close(sock);

return 0;
}
```

❸

❹

送信側サンプルプログラムは、実行時に接続先と送信するファイルパスを指定します(❶)。たとえば、送信側サンプルプログラムがa.outというファイル名の場合、以下のように実行します。

```
./a.out 10.5.6.7 hoge.txt
```


この実行例では、10.5.6.7という宛先にhoge.txtというファイルを送信しています。10.5.6.7はIPアドレスではなく、FQDNでも大丈夫です。

❷の部分は、ファイルを読み込み用に開いています。その後、❸でファイル受信側と接続し、❹でファイルを読み込みながら送信しています。❹のwhileループは、ファイルの終端まで読み込みが終わり、read ()がファイルの終わり(EOF)を意味する0を返すか、エラーによって-1を返すまで繰り返されます。

whileループを抜けると、ファイル送信側プログラムはソケットを閉じて終了します。

ファイル受信側サンプルプログラム

次は、ファイルを受信する側のサンプルプログラムです。

こちらはlisten ()とaccept ()を行うことで、ファイル送信側からのconnect ()に対応しています。

List 2-24　ファイル受信側

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <fcntl.h>

int
main(int argc, char *argv[])
{
  int sock0;
  struct sockaddr_in client;
  socklen_t len;
  int sock;
  struct addrinfo hints, *res;
  int err;
  int fd;
  int n, ret;
  char buf[65536];

  if (argc != 2) {
    fprintf(stderr, "Usage : %s outputfilename\n", argv[0]);
    return 1;
  }

  fd = open(argv[1], O_WRONLY | O_CREAT, 0600);
  if (fd < 0) {
    perror("open");
```

❶

```
  return 1;
}

memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET;
hints.ai_flags = AI_PASSIVE;
hints.ai_socktype = SOCK_STREAM;
err = getaddrinfo(NULL, "12345", &hints, &res);
if (err != 0) {
  printf("getaddrinfo : %s\n", gai_strerror(err));
  return 1;
}

/* ソケットの作成 */
sock0 = socket(res->ai_family, res->ai_socktype, 0);
if (sock0 < 0) {
  perror("socket");
  return 1;
}

if (bind(sock0, res->ai_addr, res->ai_addrlen) != 0) {
  perror("bind");
  return 1;
}

freeaddrinfo(res); /* addrinfo構造体を解放 */

/* TCPクライアントからの接続要求を待てる状態にする */
listen(sock0, 5);

/* TCPクライアントからの接続要求を受け付ける */
len = sizeof(client);
sock = accept(sock0, (struct sockaddr *)&client, &len);
if (sock < 0) {
  perror("accept");
  return 1;
}

while ((n = read(sock, buf, sizeof(buf))) > 0) {
  ret = write(fd, buf, n);
  if (ret < 1) {
    perror("write");
    break;
  }
}

/* TCPセッションの終了 */
close(sock);

/* listen するsocketの終了 */
close(sock0);

return 0;
}
```

❷

❸

受信側サンプルプログラムは、実行時に受信したファイルを保存するファイルパスを指定します（❶）。指定されたファイルパスにファイルが存在していなければ新たにファイルが作成され、ファイルのパーミッションは作成者のみが読み書きできるものになります。

たとえば、受信側サンプルプログラムがa.outというファイル名の場合、以下のように実行します。

```
./a.out hogesave.txt
```

この実行例では、受信したファイルをhogesave.txtというファイルとして保存しています。

❷の部分は、ネットワークからファイルデータを受信するためのソケットを用意し、bind ()、listen ()、accept () を行っています。

❷の部分でTCP接続を確立したあとに、❸ではネットワークからデータを読み込みながらファイルへと書き込んでいます。❸のwhileループは、送信側がファイルデータをすべて送信し終わってclose () を行い、read () がEOFを意味する0を返すか、エラーによって-1を返すまで繰り返されます。

whileループを抜けると、ファイル受信側プログラムはソケットを閉じて終了します。このサンプルプログラムは複数回accept () するようには実装されておらず、ひとつのTCP接続が終了するとともにプロセスも終了します。

ここで紹介したファイル転送プログラムは、ファイルの中身のみを転送しています。ファイル名や、ファイルパーミッションなどの付属情報も転送するには、何らかのプロトコルを規定することによって、送信側から受信側にそれらの情報を伝えなければなりません。

次に紹介するHTTPでは、最初にヘッダ情報が送信されたあとにデータ本体が送信されるプロトコルになっています。このように、データ本体と付属情報という視点で通信プロトコルをみていくと、いろいろと面白い発見があると思います。

2-6 単純なHTTPクライアント/サーバ

Chapter 2のまとめとして、より身近なプログラムに近いものを実装してみます。インターネットといえば、Webとメールでの利用が多いでしょう。ここでは、非常に単純化したWebクライアント（HTTPクライアント）とWebサーバ（HTTPサーバ）を作成し、「通信ってこんな感じなんだ」という実感を持っていただければと思います。

HTTP クライアントの実装

今度は、クライアントの例として単純なHTTPクライアントを先に作ります。HTTPは日ごろよく使っていて、なじみ深いでしょう。ただし、ここで実装するのはHTTPメッセージを表示するだけの簡単なものです。

List 2-25　単純な HTTP クライアント

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <errno.h>

int
main(int argc, char *argv[])
{
  int err;
  int sock;
  char buf[32];
  char *deststr;
  struct addrinfo hints, *res0, *res;

  if (argc != 2) {                                                    ❶
    printf("Usage : %s dest\n", argv[0]);
    return 1;
  }
  deststr = argv[1];

  memset(&hints, 0, sizeof(hints));
  hints.ai_socktype = SOCK_STREAM;                                    ❷
  hints.ai_family = PF_UNSPEC;

  if ((err = getaddrinfo(deststr, "http", &hints, &res0)) != 0) {     ❸
    printf("ERROR : %s\n", gai_strerror(errno));
    return 1;
  }

  for (res=res0; res!=NULL; res=res->ai_next) {
    printf("%d\n", res->ai_family);
    sock = socket(res->ai_family, res->ai_socktype, res->ai_protocol); ❹
    if (sock < 0) {
      continue;
    }

    if (connect(sock, res->ai_addr, res->ai_addrlen) != 0) {
      close(sock);
      continue;
    }

    break;
  }

  freeaddrinfo(res0);                                                 ❺

  if (res == NULL) {                                                  ❻
    /* 有効な接続ができなかった */
    fprintf(stderr, "failed\\n");

    return 1;
  }

  /* HTTPで「/」をリクエストする文字列を生成 */
  snprintf(buf, sizeof(buf), "GET / HTTP/1.0\r\n\r\n");

  /* HTTPリクエスト送信 */
  int n = write(sock, buf, (int)strlen(buf));                         ❼
  if (n < 0) {
    perror("write");
    return 1;
  }

  /* サーバからのHTTPメッセージ受信 */
  while (n > 0) {
    n = read(sock, buf, sizeof(buf));
    if (n < 0) {
      perror("read");
      return 1;
    }

    /* 受信結果を標準出力へ表示(ファイルディスクリプタ1は標準出力) */
    write(fileno(stdout), buf, n);                                    ❽
  }

  close(sock);                                                        ❾

  return 0;
}
```

プログラムの流れは以下のようになっています。

まず、引数の個数をチェックしています(❶)。プログラム実行時の第一引数をWebサーバのFQDN(Fully Qualified Domain Name)[注2-6]として利用しています。次に、getaddrinfo()関数に渡すためのaddrinfo構造体を設定しています(❷)。あわせてHTTP(TCPの80番ポート)用のsockaddrを結果として返すように設定し、getaddrinfo()関数を実行しています(❸)。

getaddrinfo()関数の結果に対して、ひとつずつconnect()を試みます(❹)。getaddrinfo()

の結果はIPv4とIPv6の場合がありえますが、ソケットを作成するときにはIPv4かIPv6を指定しなければならないため、socket()システムコールをfor文のなかに書いてあります。connect()に失敗した場合は、作成したソケットを閉じます。ここでは毎回ソケットを作成していますが、IPv4用とIPv6用の2つのソケットをあらかじめ作成するという方法もあります。

❺では、getaddrinfo()関数によって確保されたメモリ領域を解放しています。

変数resがNULLのときにfor文を抜けます(❻)。これは、getaddrinfo()関数の結果すべてを試し、ひとつもconnect()に成功しなかったことを示しています。そのため、エラーを表示してプログラムを終了しています。

❼は送信処理です。まずHTTPによるリクエストを作成し、Webサーバに対して送信しています。そのあとサーバからの返答を受信して、標準出力(stdout)へ出力しています(❽)。この処理は、サーバ側がデータをすべて送信し終わってTCPコネクションを切るまで続きます。TCPコネクションを切るのはサーバ側です。

最後にソケットを閉じてプログラムを終了しています(❾)。

サンプルプログラムの動作例

せっかくなので、このHTTPクライアントを使ってみたいと思います。www.google.co.jpに接続すると以下のようになります(表示スペースの関係上、一部結果を削ってあります)。

```
% ./a.out www.google.co.jp
HTTP/1.0 302 Found
Location: http://www.google.co.jp/cxfer?c=PREF%3D:TM%3D11O5:S%3DrI6jLhtK7m&
prev=/
Set-Cookie: PREF=ID=04f0a2d0a340O510:CR=1:TM=11O515:LM=11O5515:S=55PyTj
S-5evTiH; expires=Sun, 17-Jan-2038 19:14:07 GMT; path=/; domain=.google.com
Content-Type: text/html
Server: GWS/2.1
Content-Length: 217
Connection: Keep-Alive

<HTML><HEAD><TITLE>302 Moved</TITLE></HEAD><BODY>
<H1>302 Moved</H1>
The document has moved
<A HREF="http://www.google.co.jp/cxfer?c=PREF%3D:TM%315:S%3D6Gm&prev=/">here</
A>.
</BODY></HTML>
```

注2-6：FQDNとは、「www.example.com」のように記述されたホストを示す名前を表しています。単に「ドメイン名」といったときには「example.com」というドメイン全体を表すことから、それと分けて明示的にするためにFQDNといわれます。

HTTP サーバの実装

では、クライアントにデータを送るサーバとして、単純なHTTPサーバを作ってみたいと思います。HTTPサーバはお手元のWebブラウザと接続できるので、サーバを作る感覚がわかりやすいと思います。

List 2-26　単純な HTTP サーバプログラム

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

int
main()
{
  int sock0;
  struct sockaddr_in client;
  socklen_t len;
  int sock;
  int yes = 1;
  struct addrinfo *res, hints;
  int err;
  char buf[2048];
  int n;
  char inbuf[2048];

  hints.ai_family = AF_INET;
  hints.ai_flags = AI_PASSIVE;                                    ❶
  hints.ai_socktype = SOCK_STREAM;

  err = getaddrinfo(NULL, "12345", &hints, &res);
  if (err != 0) {
    printf("getaddrinfo : %s\n", gai_strerror(err));
    return 1;
  }

  sock0 = socket(res->ai_family, res->ai_socktype, 0);            ❷
  if (sock0 < 0) {
    perror("socket");
    return 1;
  }

  setsockopt(sock0,
    SOL_SOCKET, SO_REUSEADDR, (const char *)&yes, sizeof(yes));   ❸
```

```
if (bind(sock0, res->ai_addr, res->ai_addrlen) != 0) {
  perror("bind");
  return 1;
}

if (listen(sock0, 5) != 0) {                                    ❹
  perror("listen");
  return 1;
}

// 応答用HTTPメッセージ作成
snprintf(buf, sizeof(buf),
  "HTTP/1.0 200 OK\r\n"
  "Content-Length: 20\r\n"                                      ❺
  "Content-Type: text/html\r\n"
  "\r\n"
  "HELLO\r\n");

while (1) {                                                     ❻
  len = sizeof(client);
  sock = accept(sock0, (struct sockaddr *)&client, &len);       ❼
  if (sock < 0) {
    perror("accept");
    break;
  }

  n = read(sock, inbuf, sizeof(inbuf));                         ❽

  // 本来ならばクライアントからの要求内容をパースすべきです
  write(fileno(stdout), inbuf, n);

  // 相手が何をいおうとダミーHTTPメッセージ送信
  write(sock, buf, (int)strlen(buf));                           ❾

  close(sock);                                                  ❿
 }

 close(sock0);                                                  ⓫

 return 0;
}
```

最初に、TCPの待ち受けポートに関するパラメータ設定を行います(❶)。このパラメータを基にgetaddrinfo () をAI_PASSIVEで利用しています。

次に、TCPのコネクションを受け付けるためのソケットを作成しています(❷)。socket () システムコールのパラメータとして、getaddrinfo () の結果を利用していますが、getaddrinfo () へのパラメータとしてAF_INETを指定しているため、実際にはIPv4のみを受け付けるプログラムとなっています。

続いてSO_REUSEADDRを使ってTCPのポートを再利用できるようにしています(❸)。これによって、プログラムを繰り返し実行/終了しても困らなくなります(詳細はChapter 9)。

SO_REUSEADDRのあとに、ソケットに対してbind () を使って「名前を付けて」います。

ここまで準備できれば、さっそくlisten()システムコールによってTCPでの待ち受けを開始します(❹)。listen()システムコールの2つ目の引数は、アプリケーションが処理を待つために保留されるコネクションの最大数を表しています。たとえば、このプログラムコードはaccept()をしてからソケットに対してread()とwrite()が行われて、それらが終わるまで次のaccept()が行われませんが、その間にカーネルがTCP接続を保留できる最大数を、この第二引数で決定しています。このサンプルコードでは「5」となっています。

❺では、クライアントに送信するダミーデータをあらかじめ用意しています。クライアントが接続してからの処理を簡潔にするために、最初にダミーデータを生成しています。

そしてクライアントからの接続を待ち受け、TCPセッションを確立するとHTTPによってデータを受送信する処理を無限ループで繰り返します(❻)。このループではまず、クライアントからのTCPコネクションをaccept () システムコールで待ちます(❼)。accept () システムコールの第三引数はstruct sockaddrの大きさを表す変数を要求するため、このシステムコールの前にstruct sockaddr_inのサイズを変数lenに代入しています。

相手から送信されてきたHTTPリクエストデータを受信しているのが❽の部分です。本来ならば内容を解析して適切な処理を行うべきですが、このサンプルでは簡潔に保つために何もしていません。あらかじめ作成しておいたダミーデータを接続してきたHTTPクライアントに送信します(❾)。そして、accept () によって作成されたソケットを閉じています(❿)。

最後に終了処理を書いてありますが(⓫)、このプログラムは単純な無限ループとなっているため、ここまでは到達しません。

2-7 Chapter 2のまとめ

Chapter 2では、TCPを使った通信プログラミングのなかから、基本的なところを中心に解説しました。エラー処理やIPv6を意識したプログラミングなど、まず押さえておくべきポイントをまとめてあります。

次のChapter 3では、TCPと双璧をなすUDPについて、その基礎を解説していきます。

COLUMN

非同期シグナルセーフな関数

シグナルハンドラ内では、さらにシグナルが発生する可能性があるため、シグナルハンドラ内で利用できる関数には制限があります。不用意な関数をシグナルハンドラ内で利用してしまうと、思わぬバグに悩まされることもあります。

たとえば、リエントラント（呼び出されている途中に再度呼び出されることに対応できていない）関数によってデッドロックを起こしたり、データの上書きなどによって予期しない結果が発生する可能性があります。このようなバグはシグナル発生タイミングに依存することが多いため、バグを発見しにくいのも問題点のひとつです。

シグナルハンドラ内で利用しても問題が発生しない関数を、非同期シグナルセーフな関数といいます。POSIXでは、非同期シグナルセーフな関数として以下のサイトで定義しています。

URL **The Open Group Base Specifications Issue 6**
IEEE Std 1003.1, 2004 Edition , 2.4 Signal Concepts
http://www.opengroup.org/onlinepubs/009695399/functions/xsh_chap02_04.html

具体的には、以下の関数群になります。

図2-A　非同期シグナルセーフな関数

_Exit()、_exit()、abort()、accept()、access()、aio_error()、aio_return()、aio_suspend()、alarm()、bind()、cfgetispeed()、cfgetospeed()、cfsetispeed()、cfsetospeed()、chdir()、chmod()、chown()、clock_gettime()、close()、connect()、creat()、dup()、dup2()、execle()、execve()、fchmod()、fchown()、fcntl()、fdatasync()、fork()、fpathconf()、fstat()、fsync()、ftruncate()、getegid()、geteuid()、getgid()、getgroups()、getpeername()、getpgrp()、getpid()、getppid()、getsockname()、getsockopt()、getuid()、kill()、link()、listen()、lseek()、lstat()、mkdir()、mkfifo()、open()、pathconf()、pause()、pipe()、poll()、posix_trace_event()、pselect()、raise()、read()、readlink()、recv()、recvfrom()、recvmsg()、rename()、rmdir()、select()、sem_post()、send()、sendmsg()、sendto()、setgid()、setpgid()、setsid()、setsockopt()、setuid()、shutdown()、sigaction()、sigaddset()、sigdelset()、sigemptyset()、sigfillset()、sigismember()、sleep()、signal()、sigpause()、sigpending()、sigprocmask()、sigqueue()、sigset()、sigsuspend()、sockatmark()、socket()、socketpair()、stat()、symlink()、sysconf()、tcdrain()、tcflow()、tcflush()、tcgetattr()、tcgetpgrp()、tcsendbreak()、tcsetattr()、tcsetpgrp()、time()、timer_getoverrun()、timer_gettime()、timer_settime()、times()、umask()、uname()、unlink()、utime()、wait()、waitpid()、write()