C言語を用いた
ネットワークプログラミングの基礎

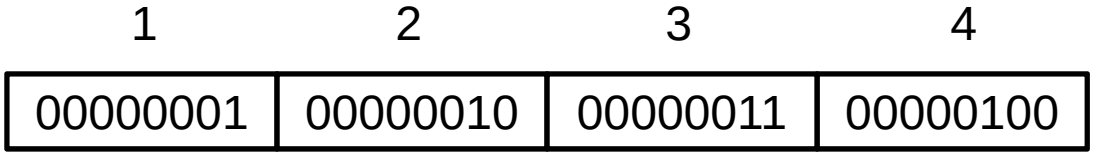
齊藤義仰

ホストバイトオーダとネットワークバイトオーダ　1/2

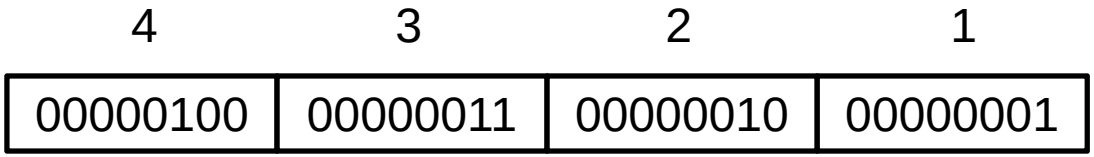
- CPUによって2バイト以上の整数データの取り扱い方（ホストバイトオーダ）が異なる
  - ビックエンディアン(Sun SPARC系プロセッサ)
    - 確保したメモリの先頭アドレスから順にデータを格納
  - リトルエンディアン（Intel x86系プロセッサ）
    - 確保したメモリの後方アドレスから順にデータを格納

（例：0x01020304の場合）

- ビックエンディアン



- リトルエンディアン



ホストバイトオーダとネットワークバイトオーダ　2/2

- ネットワーク上ではバイトオーダの統一が必要
  - ネットワークバイトオーダはビックエンディアンで統一
- 16ビット（2バイト），32ビット（4バイト）の記号なし整数データのバイトオーダを変換する関数

#include<netinet/in.h>

- uint32_t htonl(unint32_t hostlong);
  - ホストバイトオーダからネットワークバイトオーダへ（32ビット）
- uint16_t htons(uint16_t hostshort);
  - ホストバイトオーダからネットワークバイトオーダへ（16ビット）
- uint32_t ntohl(uint32_t netlong);
  - ネットワークバイトオーダからホストバイトオーダへ（32ビット）
- uint16_t ntohs(uint16_t netshort);
  - ネットワークバイトオーダからホストバイトオーダへ（16ビット）

htonl()を使ってみよう

- プログラムソース
  - 配布資料の“htonl.c”を参照
  - gccを使ってコンパイル

  gcc –o htonl htonl.c
- 実行方法
  - 実行時に引数等は必要なし

  $ htonl

- htonsを使って16ビットの場合も試してみよう
- htonlを使ってネットワークバイトオーダにしたものを
  ntohlを使ってホストバイトオーダに再変換しよう

ホスト名からIPアドレスを取得する方法

- ホスト情報を格納するhostent構造体

```
struct hostent {
    char *h_name;         // ホストの正式名称
    char **h_aliases;     // エイリアス名のリスト
    int h_addrtype;       // ホストアドレスのタイプ（AF_INET/AF_INET6）
    int h_length;         // アドレス長
    char **h_addr_list;   // アドレスのリスト（ネットワークバイト順のバイナリ値）
};
```

- ホスト名からIPアドレスを取得する関数

```
#include <netdb.h>
struct hostent  *gethostbyname(const char *hostname)
```

gethostbyname()を使ってみよう

- プログラムソース
  - 配布資料の“gethostbyname.c”を参照
  - gccを使ってコンパイル

  gcc -o gethostbyname gethostbyname.c

- 実行方法
  - ホスト名を引数に与える

  $ gethostbyname www.iwate-pu.ac.jp

- 補足情報
  - 下記の関数を使うともっと簡単にIPアドレスを表示可能

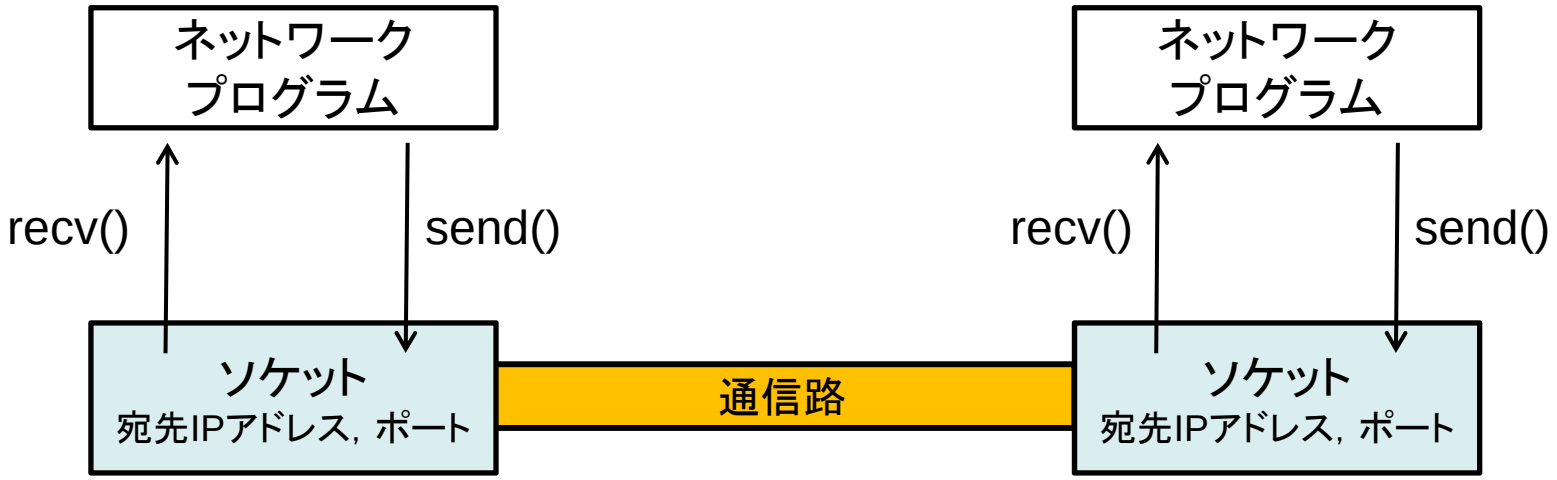
  char *inet_ntoa(struct in_addr in);

  - IPアドレスからホスト名を取得する関数も存在

  struct hostent *gethostbyaddr(const char *addr, int len, int family);

ソケット通信

- ネットワークプログラミングではSocket APIを利用して通信を行う.
- ソケットとはいわば通信のための出入り口
- ソケットには宛先のIPアドレスとポート番号を指定



クライアントでソケットを利用するまでの流れ（TCP）

1. ソケットを作成する　**socket()**

2. **sockaddr_in構造体**にアドレスファミリ，IPアドレス，ポート番号を設定

3. 自分のソケットと相手先のソケットの間でコネクションを作る **connect()**

4. ソケットを利用して入出力を行う　**recv()**，**send()**

5. 通信が終わったらソケットを閉じる　**close()**

ソケットを作成する関数：socket()

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

int socket(int protocol_family, int type, int protocol)

return value: 作成したソケットのディスクリプタ

- protocol_family
  - IPv4を利用する場合: PF_INET
  - IPv6を利用する場合: PF_INET6
- type
  - TCPを利用する場合： SOCK_STREAM
  - UDPを利用する場合： SOCK_DGRAM
  - 既存のトランスポートプロトコルを使用しない場合： SOCK_RAW
- protocol
  - TCPを利用する場合： IPPROTO_TCP
  - UDPを利用する場合： IPPTORO_UDP
  - 既存のトランスポートプロトコルを使用しない場合： IPPROTO_RAW

- 使用例（TCPソケットを作成）

```
int sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
```

インターネットアドレスファミリのアドレス構造体：sockaddr_in

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

struct  sockaddr_in {
  uint16_t  sin_family;          // TCP/IP(AF_INET)
  uint16_t  sin_port;            // ポート番号（16ビット）
  struct in_addr  sin_addr;      // IPアドレス（32ビット）
  uint8_t  sin_zero[8];          // 未使用
};

struct  in_addr {
  uint32_t  s_addr;    // IPアドレス（32ビット）
};
```

- 使用例（通信相手のアドレス情報を設定）

```
struct  sockaddr_in  sa;
memset(&sa,  0,  sizeof(sa));
sa.sin_family = AF_INET;
sa.sin_port = htons(80);
sa.sin_addr.s_addr = inet_addr(“192.168.0.1”);
```

ソケットを接続する関数：connect()　1/2

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

int connect(int socket, struct sockaddr *address, int length)

return value: 接続できれば0，失敗した場合は-1を返す

- socket
  - 接続を行うソケットのディスクリプタ
- address
  - 宛先のソケット情報を格納したsockaddr構造体へのポインタ
- length
  - 指定したsockaddr構造体のバイト長

- struct sockaddr構造体は汎用のアドレス構造体

```
#include <sys/socket.h>
struct  sockaddr {
  uint16_t  sa_family;        // アドレスファミリ（AF_INET）
  uint8_t  sa_data[14];       // プロトコル固有のアドレス情報
};
```

ソケットを接続する関数：connect()　2/2

- 使用例（192.168.0.1の80番ポートにTCPでコネクションを作りたい場合）

```
int sock;
struct  sockaddr_in  sa;

sock = socket(PF_INET,  SOCK_STREAM,  IPPROTO_TCP);

memset(&sa,  0,  sizeof(sa));
sa.sin_family = AF_INET;
sa.sin_port = htons(80);
sa.sin_addr.s_addr = inet_addr(“192.168.0.1”);

if (connect(sock,  (struct  sockaddr *) &sa,  sizeof(sa)) < 0 ) {
  fprintf(stderr, “connect() failure¥n”);
}
```

データを送信する関数：send()

#include <sys/socket.h>

int send(int socket, const void *msg, int length, int flags)

return value: 送信したバイト数．エラーの場合は-1を返す．

- socket
  - 送信に利用するソケットのディスクリプタ
- msg
  - 送信するデータへのポインタ
- length
  - 送信するバイト数
- flags
  - 制御フラグ（通常は0）

- 使用例

```
char msg[] = “Hello, World!”;
int msg_len = strnlen(msg, sizeof(msg));
send(sock, msg, msg_len, 0);
```

データを受信する関数：recv()

```
#include <sys/types.h>
#include <sys/socket.h>
int recv(int  socket,  void *buffer,  int  length,  int flags)
```

return value: 受信したバイト数．エラーの場合は-1を返す．

- socket
  - 受信に利用するソケットのディスクリプタ
- buffer
  - データを格納するバッファへのポインタ
- length
  - バッファに格納する最大バイト数
- flags
  - 制御フラグ（通常は0）

- 使用例

```
 char buf[1024];
while ((n = recv(sock, buf, sizeof(buf), 0)) > 0){
  buf[n] = '¥0';
  printf("%s", buf);
}
```

ソケットによる通信を終了させる関数：close()

#include <unistd.h>

int close(int socket)

return value: 正常終了の場合は0，エラーの場合は-1を返す．

- socket
  - 閉じるソケットのディスクリプタ（connect後でなければならない）

- 使用例

```
close(sock);
```

簡単なHTTPクライアントを作ってみよう

- プログラムソース
  - 配布資料の“test_client.c”を参照
  - gccを使ってコンパイル

  gcc -o test_client test_client.c

- 実行方法
  - ウェブサーバのホスト名とファイルパスを引数に与える
    - 例1：http://www.iwate-pu.ac.jp/ にアクセスする場合

      $ test_client　www.iwate-pu.ac.jp　/
    - 例2： http://www.iwate-pu.ac.jp/about/sitemap.html にアクセスする場合

      $ test_client　www.iwate-pu.ac.jp　/about/sitemap.html

エコークライアントを作って見よう

- プログラムソース
  - 配布資料の“echo_client1.c”を参照
  - gccを使ってコンパイル

  gcc -o echo_client1 echo_client1.c

- 実行方法
  - エコーサーバのIPアドレスを引数に与える（今回は先生がエコーサーバを立ち上げます）

  ./echo_client1 172.16.162.31

サーバでソケットを利用するまでの流れ（TCP）

1. 接続要求受付用のソケットを作成する　**socket()**
2. **sockaddr_in構造体**にアドレスファミリ，IPアドレス，ポート番号を設定
3. IPアドレスとポートをソケットに関連付ける　**bind()**
4. ソケットを接続要求を受付可能な状態にする　**listen()**
5. 接続要求を待ち，通信用のソケットを作成する **accept()**
6. 通信用ソケットを利用して入出力を行う　**recv()**，**send()**
7. 通信が終わったらソケットを閉じる　**close()**

待ち受け用のソケットの作成とアドレス構造体の設定

- ソケットの作成・アドレス構造体の設定の仕方

```
int  listen_sock;
struct  sockaddr_in  sa;

listen_sock  =  socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);

memset(&sa, 0, sizeof(sa));
sa.sin_family = AF_INET
// 接続要求を受け付けるIPアドレスとポート番号を指定
sa.sin_addr.s_addr = htonl(INADDR_ANY);  // どんなIPアドレスでもOK
sa.sin_port = htons(49152);                    // 49152番ポートで待ち受ける
```

IPアドレスとポートをソケットに関連付ける関数：bind()

```
#include <sys/types.h>
#include <sys/socket.h>
int bind (int  socket,  struct  sockaddr  *address,  unsigned int  length)
```

return value:　正常終了の場合は0，エラーの場合は-1を返す

- socket
  - ソケットのディスクリプタ
- address
  - ソケットに関連付けるローカルIPアドレス，ポート番号を格納したsockaddr構造体へのポインタ
- length
  - sockaddr構造体のバイト数

- 使用例

```
// 前のページからの続きです
bind(sock, (struct sockaddr *)&sa, sizeof(sa));
```

ソケットを接続要求を受付可能にする関数：listen()

```
#include <sys/types.h>
#include <sys/socket.h>
int listen (int  socket,  int  backlog)
```

return value:　正常終了の場合は0，エラーの場合は-1を返す

- socket
  - ソケットのディスクリプタ
- backlog
  - キューに格納できる新規コネクションの最大数

- 使用例

```
// 前のページからの続きです
listen(listen_sock,  1024);
```

接続要求を待ち，通信用ソケットを作成する関数：accept()

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

int accept (int socket, struct sockaddr *address, int *length)

return value: 新規ソケットのディスクリプタを返す．エラーの場合は-1を返す．

- socket
  - listen状態になっているソケットのディスクリプタ
- address
  - 新規ソケットの接続しているIPアドレス，ポート情報を格納するsockaddr構造体へのポインタ
- length
  - sockaddr構造体のバイト数（入力），返されたアドレスの長さ（出力）

- 使用例

```
// 前のページからの続きです
while(1){
  struct sockaddr_in new_sa;
   int len = sizeof(new_sa);
  int comm_sock = accept(listen_sock, (struct sockaddr *)&new_sa, &len);
}
```

accept()後の通信開始から終了まで

- accept()後の通信の仕方

```
// 前のページの拡張です
 while(1){
    int n;
    char  buf[1024];
    struct sockaddr_in new_sa;
    int len = sizeof(new_sa);
    int comm_sock = accept(listen_sock, (struct sockaddr *)&new_sa, &len);
    while ((n = recv(comm_sock, buf, sizeof(buf), 0)) > 0){
      buf[n] = '¥0';
      printf("%s", buf);
    }
    close(comm_sock);
 }
```

繰り返し型のエコーサーバを作ってみよう

- プログラムソース
  - 配布資料の “echo_server1.c”を参照
  - gccを使ってコンパイル
- 実行方法
  - まずサーバを引数なしで起動

```
$ echo_server1
```

  - 次に新しいターミナルをたちあげて，以前作成したecho_client1をサーバのアドレス（つまり自分自身なのでループバックアドレスである127.0.0.1）を引数に起動

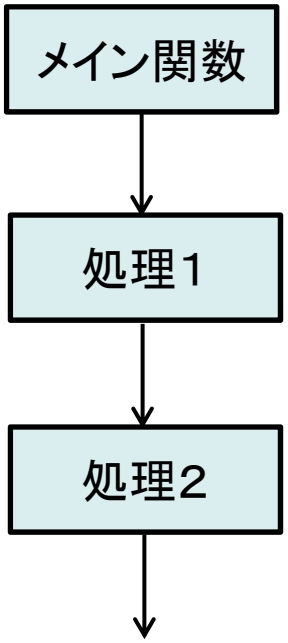
```
$ echo_client1 127.0.0.1
```

  - クライアントが入力を求めてくるので適当な文字列を打ち込むと，サーバから同じ文字列が返ってくる
  - 同時に複数クライアントを起動するとどうなるか試そう

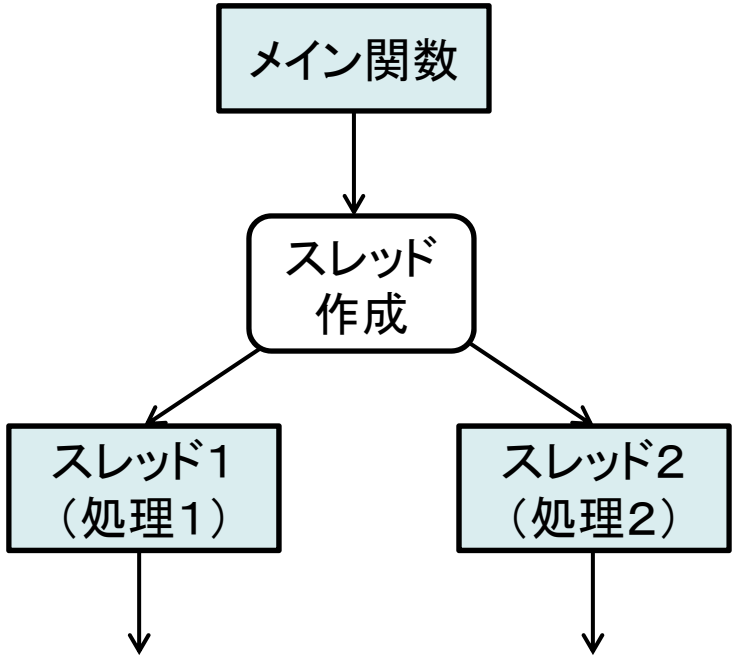
複数のクライアントを同時に処理するために

- 前のページで実装したエコーサーバでは，複数のクライアントを同時に処理することができない！
- そんなときのためにスレッドという仕組みがある
  - スレッドを使えば複数の処理を同時実行することが可能

いままでのプログラミング



スレッドを使ったプログラミング



スレッドを利用するための関数群

- POSIXスレッド（Pthread）

#include <pthread.h>

スレッドを作る

```
int pthread_create(pthread_t  *thread, const pthread_attr_t  *attr,
                   void *(*start_routine)(void *), void  *arg);
```

スレッドが終了するまで待つ

```
int pthread_join(pthread_t  thread, void  **thread_return);
```

スレッドをデタッチする（終了したらメモリ開放できるように設定）

```
int pthread_detach(pthread_t  thread);
```

呼び出しスレッドのIDを返す

```
int pthread_self(void);
```

スレッドを実際に利用して覚えよう

- プログラムソース
  - 配布資料の“pthread.c”を参照
  - gccを使ってコンパイル

  ＊注意　コンパイル時に -lpthread オプションをつける

  gcc –o pthread pthread.c -lpthread

- 実行方法
  - 引数なしで起動

  $ ./pthread

エコーサーバを並行型通信ができるようにしよう

- これをレポート課題（齊藤担当分）とします！
  - 提出物
    - 学籍番号と名前を書いた表紙
    - プログラムソース（自分が追加したソースコードの部分には，一行一行全てに解説文をコメントとして記述）
    - 複数クライアントの同時処理が確認できる実行結果
  - 締め切り　6/14（水）17:00まで
  - 提出先　情報ネットワーク実践論のレポートBOX（ソフトウェア演習等を出していた所に用意しました）
- プログラムソース
  - 配布資料の“echo_server2.c”とそのコメントを参考にプログラミングをしてください

＊注意　コンパイル時に -lpthread オプションをつける