情報工学実験 C ネットワークプログラミング

氏名: 山田　敬汰 (Yamada，Keita)
学生番号: 09430559

提出日: 2020 年 12 月 18 日
締切日: 2021 年 1 月 12 日

1　クライアント・サーバモデルでのデータ通信

今回の実験では，分散システムの基本的な形式であるクライアントサーバモデルを理解するために，TCP/IP ， UDP/IP で通信を行うクライアントサーバモデルのプログラムを作成した．

ここでは，プログラムの基礎の部分にあたる，クライアントとサーバ間での通信がどのような手順で行われているかについて詳しく解説する．

まず初めに，クライアントサーバモデルにおける通信の概要について説明する．クライアントサーバモデルでは，クライアント側の計算機のプロセスが，サーバ側のプロセスに対してメッセージを送信し，サーバ側が受け取ったメッセージを解釈し，適切な応答を返すことで通信を成立させている．ここでいう「メッセージ」とはクライアントとサーバの間で予め決めておいた規約（プロトコル）に沿って構築されたテキストである．（例: http プロトコルでは [メソッド名] [エンドポイント] [改行コード] の順に入力する）

次に，クライアントとサーバが通信する際の具体的な手順について時系列順に説明する．

まず，クライアント側での通信手順は以下の通りである．（通信相手のドメイン名は既知とする）

1. DNS サーバに問い合わせを送り，通信相手のドメイン名に対応する IP アドレスの値を取得する．
2. 通信相手（サーバ）と情報をやり取りするためのソケット（ファイルディスクリプタ）を作成する．
3. 通信相手（サーバ）との接続を確立する．（クライアント側のソケットとサーバ側のソケットとの対応づけを行う）
4. プロトコルに沿ったメッセージを構築し，サーバにメッセージを送信する．
5. サーバからの応答を待機する．
6. サーバから送られてきたメッセージを受信する．
7. 通信に使用したソケットを削除する．

そして，サーバ側での通信手順は以下の通りである．

1. 通信相手（クライアント）と情報をやり取りするためのソケットを作成する．
2. ソケットにメタデータを付与する．（どのポートで待ち受けるか，どの IP アドレスと接続するのか等）
3. ソケットの監視を OS に要求する．（作成したソケットに対する接続要求が行われることを OS に対して通知する）
4. クライアントからの接続要求を受け入れる．
5. クライアントから送られてきたメッセージを受信する．
6. 受信したメッセージに対して何らかの処理を行う．
7. プロトコルに沿ったメッセージを構築し，クライアントにメッセージを送信する．

2　プログラムの作成方針

ここでは，今回作成したプログラムの作成方針について，クライアントサイドとサーバサイドに分けて解説する．

2.1　名簿管理プログラム（クライアントサイド）の作成方針

クライアント側のプログラムをおおよそ以下の部分から作成するようにした．以下に，それぞれについての作成方針について述べる．

1. サーバとの通信部
2. 標準入力の解析部
3. コマンド処理部

2.1.1　サーバとの通信部

サーバとの通信部では，前章で説明したクライアント側での通信手順を，システムコールを呼び出すことによって実現する．なお，今回のプログラムでは，通信部分の一連の流れをリクエスト文字列とレスポンス文字列（を格納するポインタ）を引数とする関数として実装し，プログラム内から参照しやすいようにする．

2.1.2　標準入力の解析部

標準入力の解析部では，キーボードやファイルから入力された一行分の入力を解析し，次にどの処理を行うべきかの条件分岐を行う．当初は，クライアント側では解析を行わず入力をそのままサーバーに向けて送信する，という実装で実現しようとしていたが，クライアントの終了コマンド（%Q）等，サーバ側に送らずともコマンドに該当する処理が完了する場合，無意味な通信をするこ

とになるため，クライアント側でも入力の解析を行うようにした．

2.1.3　コマンド処理部

コマンド処理部では，解析した入力の先頭文字が % だった場合に，その直後の文字に応じた処理を行っている．今回のプログラムでは %Q，%C，%P，%R，%W，%H の 6 つのコマンドを実装した．これらのコマンドの詳しい動作については後述する．

2.2　名簿管理プログラム（サーバサイド）の作成方針

サーバ側のプログラムをおおよそ以下の部分から作成するようにした．以下に，それぞれについての作成方針について述べる．

1. クライアントとの通信部
2. リクエスト解析部
3. コマンド処理部

2.2.1　クライアントとの通信部

クライアントとの通信部では，前節で説明したサーバ側での通信手順を，システムコールを呼び出すことによって実現する．今回の名簿管理プログラムの場合は，一行ずつ入力を待つループの部分が，クライアント側からのメッセージ受信を待つループに置き換わっていると言える．

2.2.2　リクエスト解析部

リクエスト解析部では，クライアントから送られてきたメッセージを解析し，条件分岐を行う．クライアント側でも解析を行っているにも関わらず，サーバ側で再度解析を行う必要があるのは，今回の名簿管理プログラムにおける通信プロトコルでは，クライアント・サーバ間のやり取りが文字列で行われるからである．（メッセージを送信する際には解析情報が失われている）

2.2.3　コマンド実現部

コマンド実現部では，クライアント側と同様，解析した入力の先頭文字が % だった場合に，その直後の文字に応じた処理を行っている．なお，サーバ側では %C，%P，%W，%H の 4 種類のコマンドを実行することが可能である．これらのコマンドの詳細については後述する．

3　プログラムの説明

ここでは，前章で説明したプログラムの作成方針における分類に基づいて，それらの具体的な実装について説明する．

3.1 名簿管理プログラム（クライアントサイド）の説明

3.1.1 サーバとの通信部

サーバとの通信を以下に示す処理を実装することによって実現する．また，該当部分のソースコードは 7.1 節に添付する．

1. gethostbyname 関数を呼び出し，通信先の IP アドレスを取得する．今回の課題では，自分の PC 内でクライアントプログラムとサーバプログラムを同時に起動して実験を行うという前提のもと，ドメイン名は localhost で固定している．（ループバックアドレスである 127.0.0.1 が得られる．）
2. socket 関数を呼び出し，通信用のファイルディスクリプタ番号を取得する．
3. connect 関数を呼び出し，サーバとの接続を確立する．なお，ここで引数として渡す sa 構造体には，ポート番号や IP アドレスなどの接続先に関する情報を格納している．
4. send 関数を呼び出し，サーバへメッセージを送信する．ここでは，関数の呼び出し側がメッセージ長を気にしなくてもいいようにするために，送信したバイト数を確認し，それが BUF_SIZE と一致する限り送信を繰り返すようにしている．（BUF_SIZE で分割して送信するようにしている．）
5. receive 関数を呼び出し，サーバから送られてきたメッセージを受信する．こちらも，受信するメッセージ長が BUF_SIZE を超えても良いように，受信したバイト数が BUF_SIZE と一致する限り受信を繰り返すようにしている．

3.1.2 標準入力の解析部

標準入力の解析部では，これまでの名簿管理プログラム同様，標準入力から一行ずつ入力を読み取り，名簿データが入力された場合はそれをサーバにそのまま送信し，先頭文字が % だった場合は次の文字を見て該当するコマンド処理への条件分岐を行っている．なお，この部分のプログラムは 7.2 に添付する．

3.1.3 コマンド処理部

コマンド処理部では，先ほどの解析部での条件分岐に応じて，様々な処理を実行する．今回の名簿管理プログラムでは以下に示すコマンド処理を実装した．なお，該当の部分のソースコードを 7.3 節に添付する．

- Q コマンドが呼び出された場合，exit 関数を実行しプログラムを終了する．
- C コマンドが呼び出された場合，サーバ側に C コマンドを送信し，サーバ側から名簿の件数を取得し表示する．
- R コマンドが呼び出された場合，クライアントの計算機内に存在するファイルの内容を読

み取り，サーバ側に新規登録用の名簿データとして一行ごとに送信している．ここで，クライアント側のファイルを読み込むようにしたのは，サーバクライアントモデルでは，一つのサーバに対してクライアントが複数存在することが一般的であり，各クライアントからデータを収集するというユースケースを想定したからである．

- W コマンドが呼び出された場合，サーバ側に W コマンドを送信し，サーバ側から名簿データを前件取得しクライアント側に存在するファイルに CSV データとして書き込む．ここで，クライアント側のファイルに対する書き込みを行うようにしているのは，R コマンドの説明でも述べたとおり，一つのサーバに対してクライアントが複数存在するという状況を想定したからである．（データの永続化の役割をサーバ側が担っている）
- P コマンドが呼び出された場合，サーバ側に P コマンドを送信し，サーバ側から指定した件数のデータを受け取りプロンプトに表示する．なお，サーバからの応答の形式に違いこそあれ，「サーバ側から複数件のデータを取得し処理する」という見方をした場合，W コマンドと似たような処理をしているとも言える．
- H コマンドが呼び出された場合，サーバ側に H コマンドを送信し，サーバ側からコマンドの入力履歴のうち最新 10 件を取得し表示する．このコマンドを実装した理由は，一つのクライアントだけではサーバ側で行われた操作の全容を把握することができなかったからである．（つまり，このコマンドによってクライアント A が，クライアント B がサーバに対して行った操作を知ることができる）

3.2 名簿管理プログラム（サーバ側）の説明

3.2.1 クライアントとの通信部

クライアントとの通信を以下に示す処理を実装することによって実現する．また，該当部分のソースコードは 7.4 節に添付する．

1. `socket` 関数を呼び出し，クライアントとの通信用のファイルディスクリプタ（ソケット）を作成する．
2. `bind` 関数を呼び出し，先ほど作成したソケットとサーバ側の特定のポートを紐付ける．
3. `listen` 関数を呼び出し，OS に対してソケットに接続要求がなされることを通知する．
4. クライアント側からの接続要求があるまで待機し，接続要求（クライアント側の `connect` 関数）が来たタイミングで `accept` 関数を呼び出し，接続を確立する．
5. クライアント側からメッセージが送信されるまで待機し，メッセージが送られてきたタイミングで `recv` 関数を呼び出しクライアント側からのメッセージを配列に格納する．送られてきたメッセージのサイズがバッファサイズよりも大きい場合に備えて，クライアント側同様，送られてきたデータサイズに応じて `recv` を複数回繰り返すようにしている．
6. 送られてきたメッセージの解析等を行う．（詳しくは後述）
7. `send` 関数を呼び出し，クライアント側に応答メッセージを返す．こちらも，クライアント

側の send 部分と同じように送信するメッセージのサイズに応じて複数回 send を繰り返す実装とした．

3.2.2　リクエスト解析部

リクエスト解析部では，クライアントから送られてきたメッセージを解析し，新規名簿データの追加や，各コマンド処理に対する条件分岐を行っている．去年作った名簿管理プログラムと大きく異なる点としては，H コマンドの存在により，コマンドの入力があった場合にそれを入力履歴として配列に保存するようにしている．（最新の数件を保存するキュー構造のリスト）　なお，該当の部分のソースコードは 7.5 節に添付する．

3.2.3　コマンド実現部

コマンド処理部では，先ほどの解析部での条件分岐に応じて，様々な処理を実行する．今回の名簿管理プログラムでは以下に示すコマンド処理を実装した．また，該当部分のソースコードを 7.6 節に添付する．

- C コマンドが呼び出された場合，名簿の件数をクライアント側への応答として返す．
- W コマンドが呼び出された場合，保存されている名簿データを CSV 形式のテキストにフォーマットし，クライアント側への応答として返す．
- P コマンドが呼び出された場合，保存されている名簿データを与えられた引数の数によって絞り込み（上限，順序等），絞り込んだ結果をコンソール表示用のテキストにフォーマットしてクライアント側への応答として返す．
- H コマンドが呼び出された場合，リクエスト解析部で保存しておいた入力履歴の内容をクライアント側への応答として返す．

4　プログラムの使用法

今回の実験で作成した名簿管理プログラムは，クライアント側のプログラムに標準入力で CSV 形式の名簿データまたは，% から始まるコマンド入力を受け付け，場合によってはサーバ側のプログラムとも通信を行い，処理結果を標準出力に出力する．

具体例として，以下に示す入力を与えた際に得られる出力について説明する．

```
%C
5100046,The Bridge,1845-11-2,14 Seafield Road Longman Inverness,SEN Unit 2.0 Open
%C
%R data/sample.csv
%P 1
%C
```

```
%H
%Q
```

この入力データで行おうとしていることは，以下の通りである．なお，サーバ側のプログラムに関しては，クライアントの起動と同タイミングで起動したものと仮定する．

1. C コマンドを実行し，名簿の件数を表示（0 件）
2. 1 行分の名簿データを与え，サーバへ保存
3. 再び C コマンドを実行し，名簿の件数が正しく増分していることを確認（1 件）
4. R コマンドを実行し，大量の名簿データをまとめて追加
5. P コマンド（引数 1）を実行し，先頭の名簿データをサーバから取得して表示
6. C コマンドを実行し，ファイルから読み込んだデータがサーバ側に保存されていることを確認（2887 件）
7. H コマンドを実行し，これまでの操作履歴をサーバ側から取得して表示
8. Q コマンドを実行し，クライアント側のプログラムを終了

クライアント側では以下のような出力を得る．

```
0 profile(s)
1 profile(s)
success: cmd_read
Id    : 5100046
Name  : The Bridge
Birth : 1845-11-02
Addr  : 14 Seafield Road Longman Inverness
Com.  : SEN Unit 2.0 Open

2887 profile(s)
1: %C
2: %C
3: %P 1
4: %C
5: %H
```

この出力結果から，意図した操作が行われていることがわかる．

5　プログラムの作成過程における考察

ここでは，名簿管理プログラムをサーバクライアントモデルで実装する際に検討した内容，及び考察した内容について述べる．

5.1　容量の大きいデータの通信

W コマンドや P コマンドについて，当初の実装では send 関数で送れるバッファサイズの上限が存在するために，大量のテキストをクライアント側にまとめて送信することが困難であり，C コマンドで件数を取得したあと，件数分のリクエストを送り，一件ずつ名簿データを取得しクライアント側で結合するという手法を取っていた．つまり，サーバ側はコマンドとともに名簿データの番号を受け取り，該当のデータを一件を返すという実装になっていた．しかし，この実装ではクライアント側の責務が大きくなったり（P コマンドの取得範囲制御もクライアント側で行う必要がある），通信回数がデータの件数と同じになってしまい通信効率が悪い，という問題点があった．そこで，サーバ側のプログラムにおける send 関数の部分やクライアント側のプログラムにおける recv 関数の部分を改良し，通信するデータサイズが大きくなったとしても，バッファサイズで分割して必要最低限の回数で通信できるようにした．また，これらの処理は通信部に隠蔽しているため，コマンド処理側から通信するデータサイズを意識する必要がなくなり，W コマンドや P コマンドの処理をサーバ側に寄せることが可能となった．

5.2　新規実装コマンドについての考察

5.3

6　得られた結果に関する考察

6.1　不足機能に関する考察

7　作成したプログラム

7.1　名簿管理プログラムの通信部（クライアント）

```
#include "client.h"

int request(char *request, char *response) {
  struct hostent *hp;
  hp = gethostbyname("localhost");
```

```
if (hp == NULL) {
  printf("host not found\n");
  return -1;
}

// create socket
int soc = socket(AF_INET, SOCK_STREAM, 0);
if (soc == -1) {
  printf("failed to create socket\n");
  return -1;
}

struct sockaddr_in sa;
sa.sin_family = hp->h_addrtype;
sa.sin_port = htons(PORT_NO);
bzero((char *)&sa.sin_addr, sizeof(sa.sin_addr));
memcpy((char *)&sa.sin_addr, (char *)hp->h_addr, hp->h_length);

int connect_result = connect(soc,
          (struct sockaddr *)&sa, sizeof(sa));
if (connect_result == -1) {
  printf("failed to connect\n");
  return -1;
}

// send request
int send_result = 0;
int sended_bytes = 0;
do {
  send_result = send(soc, request + sended_bytes,
                     fmin(BUF_SIZE,
                     strlen(request + sended_bytes) + 1), 0);
  sended_bytes += send_result;
  if (send_result == -1) {
    perror("failed to send");
    break;
  }
```

```
        } while (send_result == BUF_SIZE);

        // receive response
        int recv_result = 0;
        sprintf(response, "");
        do {
          char tmp[BUF_SIZE] = "";
          recv_result = recv(soc, tmp, BUF_SIZE, 0);
          strcat(response, tmp);
          if (recv_result == -1) {
            printf("failed to receive\n");
            return -1;
          }
        } while (recv_result == BUF_SIZE);

        close(soc);

        return 0;
      }
```

7.2 名簿管理プログラムの標準入力解析部（クライアント）

```
#include "meibo_client.h"

int main() {
  char line[INPUT_MAX + 1];
  while (get_line_fp(stdin, line)) {
    parse_line(line);
  }
  return 1;
}

void parse_line(char *line) {
  if (*line == '%') {
    char *exec[] = {"", "", "", "", ""};
    split(line + 1, exec, ' ', 5);
```

```
    exec_command_str(exec);
  } else {
    char response[BUF_SIZE];
    request(line, response);
  }
}

void exec_command_str(char *exec[]) {
  if (!strcmp("Q", exec[0])) {
    cmd_quit();
  } else if (!strcmp("C", exec[0])) {
    cmd_check();
  } else if (!strcmp("P", exec[0])) {
    cmd_print(exec[1]);
  } else if (!strcmp("R", exec[0])) {
    cmd_read(exec[1]);
  } else if (!strcmp("W", exec[0])) {
    cmd_write(exec[1]);
  } else if (!strcmp("H", exec[0])) {
    cmd_history();
  } else {
    fprintf(stderr, "Invalid command %s: ignored.\n", exec[0]);
  }
  return;
}
```

7.3 名簿管理プログラムのコマンド処理部（クライアント）

```
#include "commands.h"

void cmd_quit() { exit(0); }

void cmd_check() {
  char response[BUF_SIZE];
  request("%C", response);
```

```
  printf("%s profile(s)\n", response);
}

void cmd_read(char *path) {
  FILE *fp;
  fp = fopen(path, "r");

  if (fp == NULL) {
    fprintf(stderr, "Could not open file: %s\n", path);
    return;
  }

  char line[INPUT_MAX + 1];
  while (get_line_fp(fp, line)) { /* fp を引数に追加 */
    char response[BUF_SIZE];
    request(line, response);
  }

  fclose(fp);

  printf("success: cmd_read\n");

  return;
};

void cmd_write(char *path) {
  FILE *fp;
  if (*path == '\0') {
    path = "data/output.csv";
  }

  fp = fopen(path, "w");
  if (fp == NULL) {
    fprintf(stderr, "Could not open file: %s\n", path);
    return;
  }
```

```
  char response[RESPONSE_BUF_SIZE];
  request("%W", response);

  fprintf(fp, response);
  fclose(fp);

  printf("write success: %s\n", path);

  return;
}

void cmd_print(char *num) {
  char query[BUF_SIZE];
  sprintf(query,"%%P %s", num);
  char response[RESPONSE_BUF_SIZE];
  request(query, response);
  printf(response);
  return;
}

void cmd_history() {
  char query[BUF_SIZE] = "%H";
  char response[BUF_SIZE];

  request("%H", response);
  printf("%s\n", response);
}
```

7.4 名簿管理プログラムの通信部（サーバー）

```
      int soc = socket(AF_INET, SOCK_STREAM, 0);
      if (soc == -1) {
        printf("failed to create socket\n");
        return 1;
      }
```

```
// bind port
struct sockaddr_in s_sa;
memset((char *)&s_sa, 0, sizeof(s_sa));
s_sa.sin_family = AF_INET;
s_sa.sin_addr.s_addr = htonl(INADDR_ANY);
s_sa.sin_port = htons(PORT_NO);
int bind_result = bind(soc, (struct sockaddr *)&s_sa, sizeof(s_sa));
if (bind_result == -1) {
  perror("failed to bind port\n");
  return 1;
}

// listen socket
int listen_result = listen(soc, 5);
if (listen_result == -1) {
  perror("failed to listen");
  return 1;
}

while (1) {
  // accept request
  struct sockaddr_in c_sa;
  int c_sa_len = sizeof(c_sa);
  int fd = accept(soc, (struct sockaddr *)&c_sa, &c_sa_len);
  if (fd == -1) {
    perror("failed to accept");
    return 1;
  }

  // receive response
  char request[BUF_SIZE] = "";
  int recv_result = 0;
  do {
    char tmp[BUF_SIZE] = "";
    recv_result = recv(fd, tmp, BUF_SIZE, 0);
    strcat(request, tmp);
    if (recv_result == -1) {
```

```
      printf("failed to receive\n");
      return -1;
    }
  } while (recv_result == BUF_SIZE);

  char response[RESPONSE_BUF_SIZE] = "";
  parse_line(request, response);

  int send_result = 0;
  int sended_bytes = 0;
  do {
    send_result =
        send(fd, response + sended_bytes,
             fmin(BUF_SIZE, strlen(response + sended_bytes) + 1), 0);
    sended_bytes += send_result;
    if (send_result == -1) {
      perror("failed to send");
      break;
    }
  } while (send_result == BUF_SIZE);

  close(fd);
}
close(soc);
```

7.5 名簿管理プログラムのリクエスト解析部（サーバ）

```
void parse_line(char *line, char *response) {
      if (*line == '%') {
        if (strlen(history[HISTORY_MAX - 1]) != 0) {
          for (int i = 0; i < HISTORY_MAX - 1; i++) {
            strcpy(history[i], history[i + 1]);
          }
          strcpy(history[HISTORY_MAX - 1], line);
        } else {
          for (int i = 0; i < HISTORY_MAX; i++) {
```

```
        if (strlen(history[i]) == 0) {
          strcpy(history[i], line);
          break;
        }
      }
    }

    char *exec[] = {"", "", "", "", ""};
    split(line + 1, exec, ' ', 5);
    exec_command_str(exec, response);
  } else {
    if (profile_data_nitems >= MAX_PROFILES) {
      sprintf(response, "The upper limit has been reached");
    } else {
      new_profile(&profile_data_store[profile_data_nitems], line);
      sprintf(response, "new profile created");
    }
  }
}

void new_profile(struct profile *p, char *line) {
  char *error;

  p->id = 0;
  strncpy(p->school_name, "", 70);
  p->create_at.y = 0;
  p->create_at.m = 0;
  p->create_at.d = 0;
  strncpy(p->place, "", 70);

  char *ret[5];
  if (split(line, ret, ',', 5) < 5) {
    printf("要素が不足しています\n");
    return;  //不都合な入力の際は処理を中断する
  }

  p->id = strtoi(ret[0], &error);
```

```
  if (*error != '\0') {
    printf("id の入力に失敗しました\n");
    return;  //数字が入力されていない場合は処理を中断する
  }

  strncpy(p->school_name, ret[1], LIMIT);

  char *date[3];
  int d[3] = {};
  if (split(ret[2], date, '-', 3) < 3) {
    printf("設立日の入力に失敗しました\n");
    return;  //不都合な入力の際は処理を中断する
  }
  int i;
  for (i = 0; i < 3; i++) {
    d[i] = strtoi(date[i], &error);
    if (*error != '\0') {
      printf("設立日の入力に失敗しました\n");
      return;  //数字が入力されていない場合は処理を中断する
    }
  }
  p->create_at.y = d[0];
  p->create_at.m = d[1];
  p->create_at.d = d[2];

  strncpy(p->place, ret[3], LIMIT);

  p->note = (char *)malloc(strlen(ret[4]) + 1);  //動的にメモリを確保
  strcpy(p->note, ret[4]);

  profile_data_nitems++;
}

void exec_command_str(char *exec[], char *response) {
  char *error;

  if (!strcmp("C", exec[0])) {
```

```
      cmd_check(response);
    } else if (!strcmp("P", exec[0])) {
      int param_num = strtoi(exec[1], &error);
      if (*error != '\0') {
        printf("パラメータは整数にしてください\n");
        return;
      }
      cmd_print(param_num, response);
    } else if (!strcmp("W", exec[0])) {
      cmd_write(response);
    } else if (!strcmp("H", exec[0])) {
      cmd_history(response);
    } else {
      fprintf(stderr, "Invalid command %s: ignored.\n", exec[0]);
    }
    return;
  }
```

7.6 名簿管理プログラムのコマンド処理部（サーバ）

```
void cmd_check(char *response) { sprintf(response, "%d", profile_data_nitems); }

void cmd_print(int p, char *response) {
  if (p > 0) {
    if (p > profile_data_nitems)
      p = profile_data_nitems;  //登録数よりも多い場合，要素数に合わせる

    int i;
    for (i = 0; i < p; i++) {
      print_profile(i, response);
    }
  } else if (p == 0) {
    int i;
    for (i = 0; i < profile_data_nitems; i++) {
      print_profile(i, response);
    }
```

```
  } else {
    if (abs(p) > profile_data_nitems)
      p = profile_data_nitems;  //登録数よりも多い場合，要素数に合わせる

    int i;
    for (i = profile_data_nitems - abs(p); i <= profile_data_nitems - 1; i++) {
      print_profile(i, response);
    }
  }
}

void print_profile(int index, char *response) {
  struct profile *p = profile_data_store_ptr[index];
  char tmp[BUF_SIZE] = "";
  sprintf(tmp, "Id    : %d\n", p->id);
  strcat(response, tmp);
  sprintf(tmp, "Name  : %s\n", p->school_name);
  strcat(response, tmp);
  sprintf(tmp, "Birth : %04d-%02d-%02d\n", p->create_at.y, p->create_at.m,
          p->create_at.d);
  strcat(response, tmp);
  sprintf(tmp, "Addr  : %s\n", p->place);
  strcat(response, tmp);
  sprintf(tmp, "Com.  : %s\n\n", p->note);
  strcat(response, tmp);
}

void cmd_write(char *response) {
  for (int i = 0; i < profile_data_nitems; i++) {
    struct profile *p = profile_data_store_ptr[i];
    char tmp[BUF_SIZE] = "";
    sprintf(tmp, "%d,%s,%04d-%d-%d,%s,%s\n", p->id, p->school_name,
            p->create_at.y, p->create_at.m, p->create_at.d, p->place, p->note);
    strcat(response, tmp);
  }
}
```

```
void cmd_history(char *response) {
  for (int i = 0; i < HISTORY_MAX; i++) {
    if (strlen(history[i]) != 0) {
      char tmp[BUF_SIZE];
      sprintf(tmp, "%d: %s\n", i + 1, history[i]);
      strcat(response, tmp);
    }
  }
}
```