情報工学実験Cレポート
クライアントサーバモデルを基にした
ネットワークプログラミング

氏名： 島谷 隼生 (Shimatani Toshiki)
学生番号: 09428526

出題日: 2018 年 12 月 04 日
提出日: 2019 年 01 月 29 日
締切日: 2019 年 01 月 29 日

1　クライアントサーバモデルとは

この章では，本実験の最終課題であるクライアントサーバモデルに基づくプログラムの作成にあたって，基礎となるクライアントサーバモデルの詳細を説明する．

1.1　クライアントサーバモデルとは

クライアント・サーバモデルとは，プログラムがそれぞれクライアント，サーバと呼ばれる 2 つの部分に機能を分割する，コンピュータネットワークにおけるソフトウェア構造の一つである．クライアントプログラムでは，他のサーバプログラムから提供されるサービスを使用する機能を，サーバプログラムでは，クライアントプログラムに対してサービスを提供する機能を持つ．

1.2　通信の仕組み

クライアントサーバモデルでは，機能を分割しているため，互いの処理の結果をネットワークを通じて交換する必要がある．そのため，クライアントプログラムとサーバプログラムでは，要求メッセージと応答メッセージという形でデータのやりとりを行っている．今回は TCP/IP 通信をメインにプログラムを作成するため，TCP/IP での送受信関数を利用してメッセージの交換を実現している．

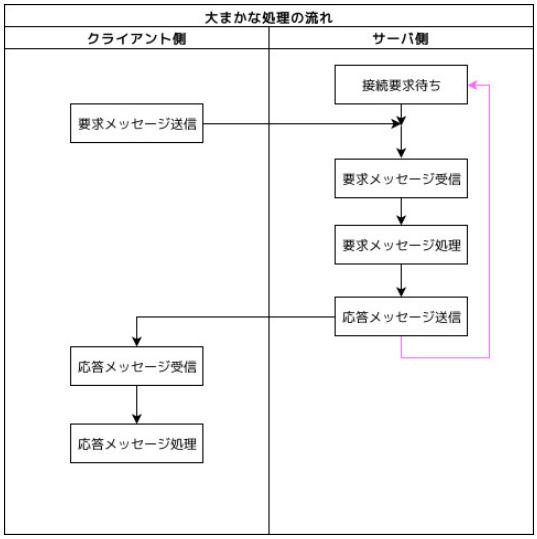
1.3　クライアントとサーバのそれぞれにおける処理の流れ

以下に大まかな処理のフローチャートを示す．

2　プログラムの作成方針

クライアントプログラムとサーバプログラムのそれぞれにおける作成方針を以下に示す．また作成にあたり使用していたフローチャートをこの章の最後に示す．

図 1: クライアントサーバモデルのフローチャート



2.1 クライアントプログラム

クライアントプログラムは，おおよそ以下の部分から構成することにした．それぞれについて作成方針を立てる．

1. プロセス間通信の前処理部（2.1.1 項)
2. 要求メッセージ送信部（2.1.2 項）
3. 応答メッセージ受信部（2.1.3 項）
4. クライアント側コマンド処理部（2.1.4 項）

2.1.1 プロセス間通信の前処理部

クライアントプログラムでの “プロセス間通信の前処理部” はクライアントサーバ間でメッセージのやりとりを行うにあたり必要な処理を行う部分である．TCP/IP を利用した通信で，クライアントがメッセージを送信するためには，まずメッセージの送り先を得る必要がある (getbyname)．その後，サーバとのデータ交換口を作成 (socket) し，サーバと接続を確立 (connect) する．ここまでがクライアントサーバ間でのメッセージ交換に必要な前処理である．

TCP/IP を利用する場合，socket 関数を使用する際に，第一引数，第二引数にそれぞれ AF_INET, SOCK_STREAM を指定する．UDP を利用したい場合は，socket 関数の第二引数を SOCK_DGRAM に変更する必要がある．

2.1.2 要求メッセージ送信部

“要求メッセージ送信部” は標準入力から得られた入力をサーバに送信する部分である．基本的には入力されたデータをそのまま送信すればよい．だがサーバ側でどのようにメッセージを読み取るのかを意識する必要がある．今回はサーバ側で一文毎にメッセージの処理を行うようにしているおり，一文の定義を改行コードがくるまでとしているため，メッセージ送信の場合に必ず終端に改行コードがある必要がある．

また，クライアント側で実行する操作がある場合にはこの段階で分岐し，コマンド処理部に処理を渡すことを推定する (具体的には%Q コマンドなど).

2.1.3　応答メッセージ受信部

“応答メッセージ受信部” はサーバから送られてきた，要求メッセージを処理した結果を受け取る部分である．基本的に受け取ったメッセージは標準出力に出力するのみであると想定する．しかしサーバ側が複数回にわけて応答メッセージを送信する場合はクライアントとサーバ間で同期をとる必要性が考えられる．

2.1.4　クライアント側コマンド処理部

“クライアント側コマンド処理部” はクライアント側で行う必要があるコマンドに対応する処理を行う部分である．具体的にはクライアントプログラムを終了する%Q コマンドやクライアントプログラムを起動した計算機にあるファイルを読み込むための%R コマンド等が想定される．

大まかな挙動は以前の実験で作成した名簿管理プログラムと共通するが，ネットワークプログラムとしていくつか変更する必要がある．例えば%Q コマンドでは，元は exit 関数を使用するだけだったが，今回はソケットをクローズする処理を追加しなければならない．

2.2　サーバプログラム

サーバプログラムは，おおよそ以下の部分から構成することにした．それぞれについて作成方針を立てる．

1. プロセス間通信の前処理部 (2.2.1 項）
2. 要求メッセージ受信部（2.2.2 項）
3. メッセージ処理部 (2.2.3 項)
4. 応答メッセージ送信部（2.2.4 項）
5. サーバ側コマンド処理部（2.2.5 項）

2.2.1　プロセス間通信の前処理部

サーバプログラムでの “プロセス間通信の前処理部” はクライアントサーバ間でメッセージのやりとりを行うにあたり必要な処理を行う部分である．クライアントとのデータの交換口を作成 (socket) する点ではクライアント側と処理は共通するが，その他は大きく異なる．サーバプログラムでは，ソケットを作成したのちにソケットに名前付け (bind) を行い，名付けしたソケットを接続待ち状態にする (listen) 必要がある．そして接続待ちとなったソケットに接続しようとしてきたクライアントを受け入れる (accept) ことでサーバプログラムの前処理は終了する．

クライアント側と同様に，TCP/IP を利用する場合，socket 関数を使用する際に，第一引数，第二引数にそれぞれ AF_INET, SOCK_STREAM を指定する．UDP を利用したい場合は，socket 関数の第二引数を SOCK_DGRAM に変更する必要がある．

また，bind 関数で使用する sockaddr 構造体のメンバ，sin_addr 構造体のメンバ s_addr に任意のアドレスを意味する INADDR_ANY を htonl 関数を使用して設定しておくことに注意する．この操作により，接続を受け付けるソケットは任意の計算機からアクセスを受け付けることができ

る．INADDR_BROADCAST を設定しても同様の効果が得られるようだが，詳細な理由は分からなかった．

accept 関数を使用する際には，引数となるソケットと返り値となるソケットが別となることに注意する．サーバ処理メインルーチンでは返り値となるソケットを用いてクライアントと通信を行う．

2.2.2 要求メッセージ受信部

“要求メッセージ受信部” ではクライアントから送信されてきたメッセージを受信する部分である．クライアント側から改行文字を区切りとする 1 行メッセージが送信される想定であるから，これを処理しやすい形に変える簡単な処理を加える．具体的にはメッセージの改行文字をナル文字の置き換えする処理である．この処理により配列処理の関数が適用できるようになり，以前作成した名簿管理プログラムの大枠を流用できると考える，

置換処理を加えた後，メッセージを “メッセージ処理部” に渡すことで，この部分の処理は終了する．

作成中は，受信したメッセージをサーバ側の標準出力に出力させることでエラーの有無を確認することを想定する．

2.2.3 メッセージ処理部

“メッセージ処理部” は “要求メッセージ処理部” で処理しやすい形に変更されたメッセージを以前作成した名簿管理プログラムで処理を行う部分である．多くの処理が流用できると考えているが，結果の出力部分は send 関数を用いてクライアント側に送信しなければならないので，多少の修正が必要である．

名簿データの登録はこの部分で行うことを想定するが，コマンド処理は “サーバ側コマンド処理部” に処理を委ねることを想定する．処理部を分割して作成することで，機能拡張が容易になると考えられる．

2.2.4 応答メッセージ送信部

“応答メッセージ送信部” は “メッセージ処理部”，“サーバ側コマンド処理部” で処理された結果をクライアント側に送信する部分である．基本的に処理内容をソケットに出力するのみであると想定する．

2.1.3 項でも述べたが，応答メッセージを繰り返して出力する必要がある場合 (%P コマンド等) は，クライアント側と同期を取る必要があると考えられる．

2.2.5 サーバ側コマンド処理部

“サーバ側コマンド処理部” はサーバ側で行う必要があるコマンドに対応する処理を行う部分である．具体的にはサーバ側に保存してある名簿データに関する情報を提供する%P コマンドや%C コマンド，ファイルに名簿データを書き込む%W コマンドが考えられる．

それぞれを関数として実装し，“メッセージ処理部” から処理を受け取れるようにする．

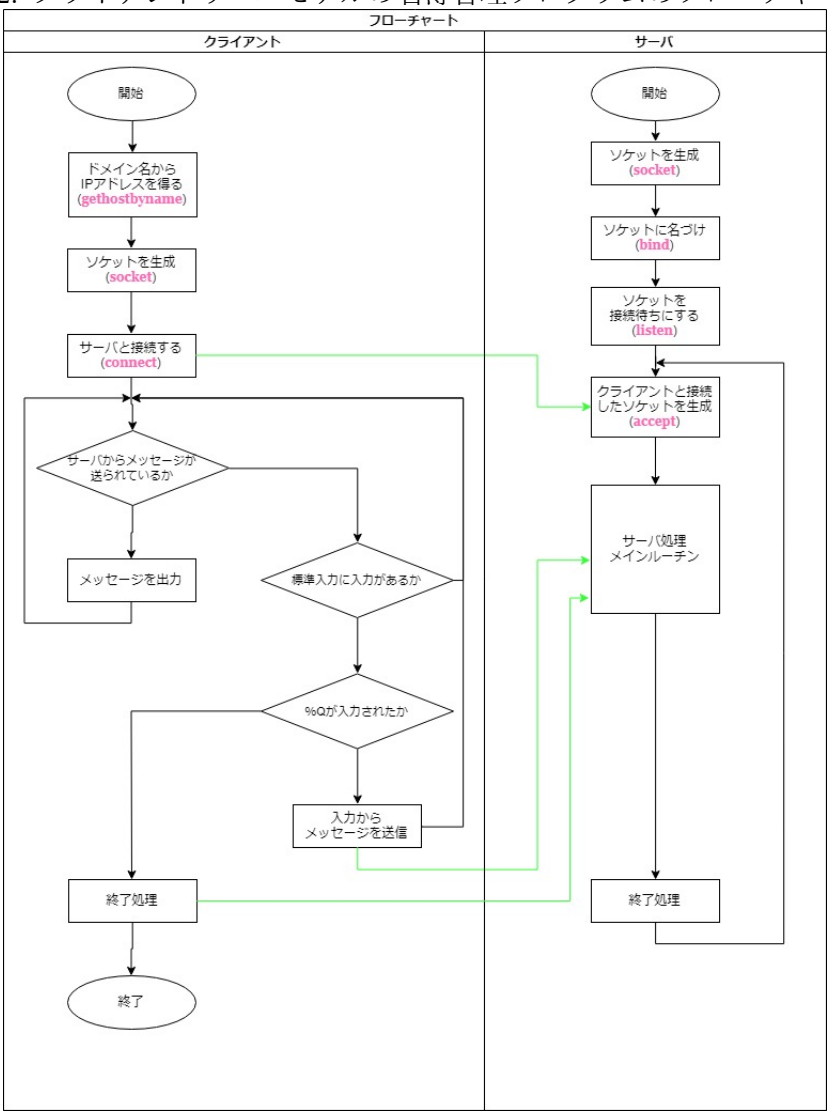
%P コマンド実装に渡り，名簿データをバッファリングし，通信処理回数を減らすことも考えられる．時間があれば実装することを考える．時間がない場合は，データを一つ一つ送信する仕様とする．

%Q コマンドに関しては，クライアントとの接続の終了処理のみにとどめ，サーバプログラム自体は終了しないように注意する．終了してしまうと，次のクライアントからの接続に応えられなくなるためである．

2.3　作成にあたり使用したフローチャート

図 2 は作成にとりかかるに当たって使用したフローチャートである．完成した名簿管理プログラムと完全に一致する訳ではないが，大まかな流れと一致するため，ここに示しておく．

図 2: クライアントサーバモデルの名簿管理プログラムのフローチャート



3　プログラムおよびその説明

プログラムリストは 7 章に添付している．クライアントプログラムとサーバプログラムは，それぞれ xx 行, yy 行からなる．作成を進めていく過程で作成方針で大まかに分類した構成要素が統

合した部分があるため，実際のプログラムの流れにしたがいながら，作成方針から修正を加えた点などを示す．

また，クライアントサーバ間でのプロトコルについてもここで説明する．

3.1 クライアントプログラム

3.1.1 プロセス間通信の前処理部

この部分では，前章で示したように，サーバと通信するために gethostbyname,socket,connect 関数を順に使用していき，要求メッセージ送信のための準備を行う．また gethostbyname 関数に渡す IP アドレス，またはドメイン名はコマンドライン引数から渡すように実装したため，プログラム実行時の第一引数に IP アドレスかドメイン名を指定する仕様となっている．引数が無い場合はエラー出力を行い，プログラムを終了させる．今回は同一計算機内で動作させることが多いため，多くの場合はコマンドラインからローカルループバックアドレスである “127.0.0.1” を指定する．

概ね方針通りに作成を行ったが，それぞれの関数でエラーが出た場合のエラー処理を加えた．エラーが出た場合は，途中でプログラムを終了する．

3.1.2 要求メッセージ送信部

この部分では，前章で示したように標準入力から得られた入力を要求メッセージとして送信する．

ここの部分も概ね方針どおりであり，改行文字が入力されるまでの入力内容を要求メッセージとして送信する．だが，作成方針でも示したように，入力内容がコマンド (%から始まる入力) であった場合，クライアント側で実行するコマンドではないかをチェックし，実行するコマンドであれば処理をコマンド関数に渡す．ここで，コマンドの内容によっては引数を持つことがあるため，二つの配列 (cmd1,cmd2) を用意し，sscanf 関数を用いて入力内容に空白が含まれる場合に分割している．

また，送信するメッセージの終端には必ず改行文字があることを定めているため，仕様に一致するように以下の記述を行っている．

```
if((n = read(0, recv_buf, BUFSIZE-1))<= 0) break;
      recv_buf[n]='\n';
```

この記述により，送信するメッセージの終端には必ず改行文字が設定される．

方針から異なり，加えた点として select 関数の使用がある．これは標準入力に一定の時間入力がなかった場合，クライアントプログラムを終了させるために実装した．詳しくは 5 章にて説明する．これに伴い，応答メッセージ受信部でも socket 関数への対応部分が加えられている．

3.1.3 応答メッセージ受信部

この部分では，前章で示したようにサーバ側から送信された応答メッセージを受信し，出力する．

方針通りに結果は出力するのみであるが，サーバ側からのメッセージの受信に失敗した場合は，エラー処理をし，プログラムを終了する．

前項でも述べたように，select 関数への対応がここにも加えられている．詳しくは 5 章にて説明する．select 関数への対応により，%P コマンドに関しては同期をとる必要がなくなった．このことも 5 章にて説明する．

3.1.4 クライアント側コマンド処理部

この部分では，前章で示したようにクライアント側で行う必要のあるコマンドを処理する部分である．

作成方針とは異なり，%Q コマンドはこの部分に記述せず，要求メッセージ送信部のコマンド処理部への分岐処理の際に，%Q に相当する処理を組み込んで実装することとした．これにより%Q コマンドの機能を果たす関数の作成は行っていない．

%R,%W コマンドは方針どおり関数を作成し，実装を行っている．前章の応答メッセージ受信部で述べたがメッセージを繰り返し送受信する場合には同期が必要となる．コマンド処理部では select 関数の恩恵が得られないため，応答メッセージ受信部とは異なり，同期が必要で，同期を行わない場合，メッセージが 2 回以上，すなわち%R,%W コマンドの場合は名簿データが 2 件以上となる場合，正しくメッセージをやりとりできなくなる．

同期は次の順に行われていく．

1. クライアントサーバ間でメッセージを送りあい，互いに準備ができたことを確認する
2. 送信側がメッセージを送信し，ループを回すことで何かメッセージが送られて来るまで待機する
3. 受信側はメッセージを受信後，処理を行い，送信側に処理が完了したことを知らせるメッセージを送信する
4. 送信側がメッセージを受け取ったら 2 に戻る

%R,%W コマンドに応じて，送信側と受信側がクライアントとサーバで入れ替わっている．また，名簿データ以外のメッセージは便宜的なものであるため，クライアント，サーバの双方で適当に ack という名前で配列を作成し，利用している．

3.2 サーバプログラム

3.2.1 プロセス間通信の前処理部

この部分では, 前章で示したように，クライアントと通信するために socket,bind,listen,accept 関数を順に使用していき，要求メッセージ受信のための準備を行う．クライアント側と同様に，概ね方針どおりに作成したが，それぞれの関数に対するエラー処理は加えている．エラー処理がでた場合はプログラムが終了するしようとなっている．

3.2.2 要求メッセージ受信部

この部分では，前章で示したように受け取った要求メッセージの末尾を置換する部分である．以下の様な実装を行っている．

```
receive: /* ストリーム型のデータの受信処理 */
  if(rn = recv(new_s,&recv_buf[i],1,0) < 0) break;
  /* 改行単位で受信処理をする */
  if (recv_buf[i] != '\n') {
    i++;
    if (i < BUFSIZE - 1)
      goto receive;
```

```
}
recv_buf[i] = '\0';
                    :
        if((recv_msg_exe(recv_buf, new_s))<=0) break;
```

ソケットに送られたメッセージを一文字ずつ読み取り，改行コードが現れれば末尾をナル文字に置き換え，メッセージ処理部に渡している．メッセージが指定のバッファサイズを超えるようであれば，途中で読み取りを終了し，末端にナル文字を置き，メッセージ処理部に渡している．この仕様により，名簿データの任意長であるコメント部がバッファサイズに制限されることとなっている．

3.2.3 メッセージ処理部

この部分では，前章で示したように，末尾が置換されたメッセージを以前作成した名簿管理プログラムで処理を行う部分である．それぞれのコマンド関数にソケットを渡す点と，名簿データの入力値が不正だった場合に標準エラー出力に出力していたエラーメッセージを応答メッセージをする点を除けば，ほとんどを流用することができた．

3.2.4 応答メッセージ送信部

この部分が最も作成方針からはずれた部分である．作成方針では独立した部分であったが，実際に作成にあたり，名簿データの登録の際は応答メッセージを返す必要がないため，コマンド処理部に応答メッセージの送信を委ねると独立して作成する必要性が低くなった．作成したプログラムで応答メッセージ送信部にあたる部分はそれぞれコマンド関数内部に含まれるか，元の名簿管理プログラムでのデータ登録の際のエラー出力の部分に統合した．

3.2.5 サーバ側コマンド処理部

この部分では，前章で示したようにサーバ側で行う必要のあるコマンドを処理する部分である．

この部分で実装したのは，%C,%P コマンドを実装した．

%P コマンド関数に関しては，元から作成してあった名簿データを出力する print_profile 関数の出力先を標準出力から送信バッファに変更することで実現した．

%C コマンドも結果の出力先を標準出力から送信バッファに変更することで実現した．

3.3 クライアントサーバ間でのプロトコル

ここではクライアントサーバ間でのプロトコルを示す．要求メッセージと応答メッセージの末尾には必ず改行文字がなければならない．この処理はユーザは特に意識する必要はないが，読み込む CSV ファイルを作成する際に，改行区切りの 1 行ずつに一つの名簿データのみ読み取ることに注意する．また以下の表では各コマンドとそのコマンド実行時の返り値を示す．

```
Id    : 8681139
Name  : Cedars School of Excellence
Birth : 1967-11-03
Addr  : Lothian Road Greenock
Com.  : 01475 631074 Primary 20 3.1 Secondary 11 2.6 Ope
```

4　プログラムの使用法

本プログラムは名簿データを管理するためのプログラムである．クライアント側で CSV 形式のデータと % で始まるコマンドを標準入力から受け付け，サーバに送信し，サーバ側で受信した内容を処理し，その結果をクライアントに送信する．

プログラムは，一般的な UNIX で用いることを意図している　gcc でコンパイルした後，プログラムを実行する．その際，サーバを先に実行することと，クライアントではコマンドライン引数で IP アドレスまたはドメイン名が必要であることに注意する．

プログラム実行後，手入力で CSV 形式でデータを入力するか，各種コマンドを使用する．

```
\$ gcc -o server server.c
\$ ./server

\$ gcc -o client client.c
\$ ./client 127.0.0.1
\$ 09428900,Takahashi Kazuyuki,1977-04-27,3,Saitama,male
\$ %C
```

プログラムの出力結果としては CSV データの各項目を読みやすい形式で出力する．例えば，下記の sample.csv に対して，

```
< 09428900,Takahashi Kazuyuki,1977-04-27,Saitama,male
< 09428901,Honma Mitsuru,1972-08-25,Hokkaidou,male
< %C
< %P
< %P 1
< %Q
```

以下のような出力を得る．

```
> 2 profile(s)

Id    : 9428590
Name  : Takahashi Kazuyuki
Birth : 1977-04-27
Addr  : Saitama
Com.  : male

Id    : 94285901
Name  : Honma Mitsuru
Birth : 1972-08-25
Addr  : Hokkaido
Com.  : male
```

表 1: 実装したコマンド

コマンド	返り値
%Q(q)	なし
%C(c)	int 型で現在のデータ登録数
%P(p) n	char 型の配列で表の上の形式のバッファ
%R(r) file	成功時は OK の文字列，失敗時はエラーメッセージ
%W(w) file	成功時は OK の文字列，失敗時はエラーメッセージ

```
Id    : 94285900
Name  : Takahashi Kazuyuki
Birth : 1977-04-27
Addr  : Saitama
Com.  : male

> Bye.
```

入力された%C は，これまでの入力データが何件登録されたかということを示し，%P は 入力したデータを全件表示することを示している．また，%P 1 は入力したデータを先頭から 1 件 (負の数だと後ろから) 表示することを示し，%Q はプログラムを終了することを示す．

上の例では%Q，%C，%P コマンドの説明を行った．以下では残りの%R，%W コマンドの説明を行う．%R，%W コマンドを使用すると，以下のような sample.csv に対して，

```
(sample.csv)
   09428900,Takahashi Kazuyuki,1977-04-27,Saitama,male
   09428901,Honma Mitsuru,1972-08-25,Hokkaidou,male
   09428902,Nakamura Hiroki,1975-09-04,Nagano,male
```

次のような応答が得られる．

```
< %R sample.csv
looding ...
OK.
< %W a.csv
OK.
< %C
> 3 profile(s)
< %P
Id    : 94285900
Name  : Takahashi Kazuyuki
Birth : 1977-04-27
Addr  : Saitama
Com.  : male

Id    : 94285901
Name  : Honma Mitsuru
Birth : 1972-08-25
Addr  : Hokkaido
Com.  : male

Id    : 94285902
Name  : Nakamura Hiroki
Birth : 1975-09-04
Addr  : Nagano
Com.  : male
```

出力後，a.csv は以下のように書き換えられている．

```
   09428900,Takahashi Kazuyuki,1977-04-27,Saitama,male
   09428901,Honma Mitsuru,1972-08-25,Hokkaidou,male
   09428902,Nakamura Hiroki,1975-09-04,Nagano,male
```

5　作成過程における考察

本章では，名簿管理プログラムの作成過程において検討した内容，工夫した内容，および，考察した内容について述べる．

5.1　%R,%W コマンドの考察

ここでは，%R と%W コマンドに関して，クライアント側かサーバ側かのどちらのファイルを指定するかを考察する．この 2 つをそれぞれ考えていく．

%R コマンドに関しては，まずその使い道を考えると，クライアント側のファイルを指定する場合ではクライアント側が保持する CSV 形式の名簿データをまとめて登録したい場合に使用することが考えられ，サーバ側のファイルを指定する場合は%W をサーバ側のファイルを指定するとした場合に，データを取り出す役割を果たす．

%W コマンドに関しては，同様に使い道を考えると，クライアント側でファイルを指定する場合は名簿データをクライアント側が手元に残したい場合に使用することが考えられる．サーバ側でのファイルを指定する場合は現在登録されているデータを一時退避させる際に使用することが考えられる．そうすることでクライアントが終了した後，再接続した際に同じデータを自動で読み取る機能の実装等に使用できる．

これらを踏まえとどちらの実装でもメリットは存在するといえる．そこで，次にクライアントサーバモデルの名簿管理プログラムという意味を考えていく．クライアントサーバモデルである以上，サーバ側の負担はできる限り少ないほうがよいと考える．つまり複数のクライアントに利用されることを想定するのであれば，サーバ側にデータを保存するとサーバ側に多くの名簿データが記憶され，場合によっては他のクライアントにデータを書き換えられる可能性がある．そのためクライアント側で%R，%W コマンドを実装するべきと考える．

5.2　サーバ側の多重受付に関する考察

当初の作成方針では，サーバは 1 つのクライアントとしか通信できず，現実的なプログラムではなかった．そのため，多数のクライアントから同時に要求を受け付けられるように拡張することを考えた．考えられる手法は，select 関数を用いる方法と，fork 関数を用いる方法である．今回採用した方法は fork 関数である．理由は 2 点あり，select 関数と fork 関数のクライアントの上限を考えた場合，select 関数はファイルディスクプタの上限数であり，fork 関数はサーバ側が同時に起動できるプロセスの最大数であるため，fork 関数の方がより多くのクライアントに対応できると考えた点と，fork 関数で実装するとプログラムを再帰的に実行させることで簡単に実現できる点である．この 2 点の理由により fork 関数を用いる手法を採用した．

この結果，サーバ側の多重受付が可能となった．

5.3　select 関数の使用

今回，クライアントプログラムで作成方針と異なった点として select 関数の使用が挙げられる．select 関数は登録したソケットを監視し，データが受信可能となったソケットに read 関数等を使用できるようにする関数である．これを実装することで，クライアントプログラムのメインルーチンを標準入力とサーバーと接続されたソケット部に分割することができ，サーバが連続してソケットにメッセージを送っても逐一同期をとる必要がなくなった．これはクライアントがコマンド

を実行したあと，サーバが繰り返しメッセージを送信しても，クライアントは受信したメッセージを処理した後，再びデータを受信しているソケットを読み込むためである．

しかし，これは select 関数のループ内に限ってである．%R や%W コマンドでコマンド関数内部で処理を行う際は，一つのソケットしかなく，入出力口が共有されているため同期が必要となる．

6　結果に関する考察

ここでは，以下の項目について考察を述べる．

1. 不足機能についての考察
2. ゾンビプロセスについての考察
3. クライアントサーバモデルによる機能の制限

6.1　不足機能についての考察

考えられる不足機能としては，登録されたデータをリセットし 0 件とする`%reset` コマンド，採用しなかったサーバ側のファイルを使用する%R,%W コマンド，実装してあるコマンドを説明する機能などが考えられる．それぞれの理由を説明する．まず，登録されたデータをリセットし 0 件とする`%reset` コマンドについてだが，データが 10000 件に達するともとの名簿管理プログラムの仕様上データの登録が出来無い．登録データを%W コマンドで書き込みした後，リセットできなければ逐一クライアントプログラムを終了する必要がある．次に，採用しなかったサーバ側のファイルを使用する%R,%W コマンドであるが，このコマンドは名簿データの共有が異なるクライアント間でできるため採用の価値があるが，%R コマンドがうまく実装できなかったため断念した．実装してあるコマンドを説明する機能は，いわゆるヘルプ機能である．プログラムを用いる人がプログラムの使用方法を完全に知っている可能性は高くないため，それを補う必要がある．そのための機能である．

6.2　ゾンビプロセスについての考察

前章で，fork 関数を利用してサーバ側の多重要求受付を実現したことを述べた．この fork 関数にはゾンビプロセスと呼ばれる問題が存在する．fork 関数の機能は子プロセスを生成するというものでる．生成された子プロセスが終了した際に親プロセスが wait 関数を用いて子プロセスの終了を確認する必要があるが，子プロセスの終了前に，親プロセスが終了してしまうと，子プロセスの終了確認ができず，プロセステーブルに残り続けるという現象が生じる．しかし wait 関数の使用中は親プロセスの処理が停止するため，名簿管理プログラムがうまくいかない．解決方法を模索したが，時間が足りず解決には至らなかった．

6.3　クライアントサーバモデルによる機能の制限

今回，名簿管理プログラムをクライアントサーバモデルに対応させたことにより，本来の仕様に制限がかかった部分がある．具体的には任意長のコメント部分である．プロセス間通信に send や recv 関数を用いるため，今回送信バッファと受信バッファを設定し，バッファサイズをマクロ定義で定めているため，定義したバッファサイズまでしかコメントを書くことができなくなってしまっている．この対策として，バッファ長以上にコメントが伸びている場合に，分割して送信することで任意長を維持できると考えたが，実装する時間がなかったため今回は実装を見送った．

7　作成したプログラム

作成したプログラムを以下に添付する.

7.1　クライアントプログラム

```
 1 #include<sys/types.h>
 2 #include<sys/socket.h>
 3 #include<netinet/in.h>
 4 #include<stdio.h>
 5 #include<netdb.h>
 6 #include<string.h>
 7 #include<stdlib.h>
 8 #include <arpa/inet.h>
 9 #include <sys/time.h>
10 #include <netinet/in.h>
11
12 #define PORT_NO 2018
13 #define BUFSIZE 1024+1
14 #define MAX_LINE_LEN 1024 /*1 行に読み込める最大文字数*/
15
16 void cmd_read(char *file, int socket);
17 void cmd_write(char *file, int socket);
18
19 int main(int argc, char* argv[]){
20   int s, i=0, len, size, n;
21   char recv_buf[BUFSIZE]={0};  /* 受信バッファ                  */
22   char send_buf[BUFSIZE]={0};  /* 送信バッファ                  */
23   char cmd1[BUFSIZE]={0};
24   char cmd2[BUFSIZE]={0};
25   struct sockaddr_in sa;
26   struct hostent *hp;
27   struct timeval tv;    /* select のタイムアウト時間       */
28   fd_set readfd;        /* select で検出するディスクリプタ  */
29   int cnt;
30   if(argc < 2){
31     fprintf(stderr,"Error: Didn't set IP_addr or domain\n");
32     exit(1);
33   }
34   if((hp = gethostbyname(argv[1]))==0){
35     fprintf(stderr,"Error: Unknwon host.\n");
36     exit(1);
37   }
38
39   sa.sin_family = AF_INET;
40   sa.sin_port = htons(PORT_NO);
41
42   bzero((char *)&sa.sin_addr, sizeof(sa.sin_addr));
43
44   memcpy((char *)&sa.sin_addr,(char *)hp->h_addr,hp->h_length);
45
46   if((s = socket(AF_INET, SOCK_STREAM, 0))==-1){
47     fprintf(stderr,"Error: Can't open socket.\n");
48     exit(1);
49   }
50   if(connect(s, (struct sockaddr*)&sa, sizeof(struct sockaddr_in))==-1){
51     fprintf(stderr,"Error: Can't connect socket with host.\n");
52     exit(1);
53   }
54
55   printf("connected to '%s'\n", inet_ntoa(sa.sin_addr));
```

```
56
57   /* client processing routine */
58   while(1){
59     //bzero(send_buf,strlen(send_buf));
60     //bzero(recv_buf,strlen(recv_buf));
61     tv.tv_sec = 600;
62     tv.tv_usec = 0;
63
64     FD_ZERO(&readfd);
65     FD_SET(0,&readfd);
66     FD_SET(s,&readfd);
67     if((select(s+1, &readfd, NULL, NULL, &tv))<=0){
68       fprintf(stderr, "\nTimeout\n");
69       break;
70     }
71
72     /* standard input */
73     if(FD_ISSET(0, &readfd)){
74       bzero(cmd1,BUFSIZE);
75       bzero(cmd2,BUFSIZE);
76       if((n = read(0, recv_buf, BUFSIZE-1))<= 0) break;
77       recv_buf[n]='\n';
78       cnt = sscanf(recv_buf, "%s%s", cmd1,cmd2);
79       if(strcmp(cmd1, "%Q") == 0 ||
80  strcmp(cmd1, "%q") == 0) {
81 printf("Bye.\n");
82 break;
83       }
84       if(strcmp(cmd1,"%W")== 0 ||
85  strcmp(cmd1,"w") == 0){
86 send(s,"%W\n",3,0);
87 if(cnt == 2) cmd_write(cmd2, s);
88 else if(cnt == 1){
89   fprintf(stderr,"%W(%w) command need argument(filename).\n");
90   fprintf(stderr,"format: %W (file name)\n");
91 }
92 continue;
93       }
94       if(strcmp(cmd1, "%R") == 0 ||
95  strcmp(cmd1, "%r") == 0) {
96 send(s,"%R\n",3,0);
97 if(cnt == 2) cmd_read(cmd2, s);
98 else if(cnt == 1){
99   fprintf(stderr,"%R(%r) command need argument(filename).\n");
100  fprintf(stderr,"format: %R (file name)\n");
101 }
102
103  continue;
104      }
105      if(send(s, recv_buf, n, 0) <= 0) break;
106    }
107
108    /* server */
109    if (FD_ISSET(s, &readfd)){
110      //recv(s,recv_buf,2,0);
111      //printf("%s",recv_buf);
112      //bzero(recv_buf,BUFSIZE);
113      if ((n = recv(s, recv_buf, (BUFSIZE)-1, 0)) < 0){
114 fprintf(stderr, "Error: connection closed. \n");
115 close(s);
116 exit(EXIT_FAILURE);
117      }
```

```
118       recv_buf[n]='\0';
119       printf("%s",recv_buf);
120       fflush(stdout);
121     }
122   }
123   bzero(send_buf, BUFSIZE);
124   strncpy(send_buf, "%Q", 2);
125   send(s, send_buf, n, 0);
126   close(s);
127
128   return EXIT_SUCCESS;
129 }
130
131
132 /*
133   while(1){
134     bzero(buf2, sizeof(buf2));
135     recv(s,buf2,5,0);
136     printf("%s",buf2);
137     bzero(buf2, sizeof(buf2));
138     scanf("%[^\n]",&buf2);
139     i=send(s, buf2, strlen(buf2)+1, 0);
140     if(i==-1){
141       close(s);
142       fprintf(stderr,"Error: Failed sending message.\n");
143       exit(1);
144     }
145     send(s, "\r\n",2,0);
146
147     recv(s, buf, strlen(buf2), 0);
148
149     printf("%s\n",buf);
150   }
151 }*/
152
153 void cmd_read(char *file, int socket) {
154   FILE *fp;
155   char line[BUFSIZE + 1];
156   char ack[2]={0};
157   int n;
158
159   fp = fopen(file, "r");
160
161   if (fp == NULL) {
162     fprintf(stderr,"Could not open file: $s\n", file);
163     return;
164   }
165
166   while(1){
167     if((recv(socket,ack,1,0))>0) {
168       printf("loading . . .\n");
169       break;
170     }
171     //printf("%d",n);
172   }
173   while (1) {
174     bzero(line,BUFSIZE+1);
175     if((fgets(line,BUFSIZE+1,fp)) == NULL) break;
176     if(strlen(line) > BUFSIZE) line[BUFSIZE] = '\n';
177     send(socket,line,BUFSIZE+1,0);
178     while(1){
179       if((recv(socket,ack,1,0)) > 0) break;
```

```
180     }
181   }
182   printf("OK.\n");
183   fclose(fp);
184   return;
185 }
186
187 void cmd_write(char *file, int socket){
188   FILE *fp;
189   char line[BUFSIZE + 1];
190   char ack[2]={0};
191   char recv_buf[BUFSIZE+1]={0};
192   int n;
193
194   fp = fopen(file, "w");
195
196   if (fp == NULL) {
197     fprintf(stderr,"Could not open file: $s\n", file);
198     return;
199   }
200
201   if(strchr(file,'/')!=NULL){
202     fprintf(stderr,"This file-name is including invalid character '/' : %s\n", file);
203     return;
204   }
205   while(1){
206     if((recv(socket,ack,1,0))>0) {
207       break;
208     }
209   }
210   send(socket," ",1,0);
211   while (1) {
212       bzero(recv_buf,BUFSIZE);
213       while(1){
214 if((recv(socket,recv_buf,BUFSIZE,0)) > 0) break;
215       }
216       recv_buf[BUFSIZE]='\0';
217       if(*recv_buf == '\0') break;
218       fprintf(fp,"%s",recv_buf);
219       send(socket," ",1,0);
220   }
221   printf("OK.\n");
222   fclose(fp);
223   return;
224 }
```