第7章　クライアント・サーバのためのプログラミング

7.1　有限状態機械に基づくプログラミング

有限状態機械 (finite state machine, FSM) あるいは**有限オートマトン (finite automaton, FA)** とは，有限個の状態，状態の遷移，動作，の組み合わせからなるモデルである．図 7.1 に有限状態機械の例を示す．図において四角は状態を示す．2 つの状態間を結ぶ線は状態の遷移を表す．線に付けられたラベルにおいて，平行線の上部は入力またはイベントを表し，下部は状態遷移に伴う動作を表す．イベントの発生に伴って状態が遷移していくので，状態はそれまでに発生したイベントの記憶を持つこととなる．

たとえば図 7.1 において，現在プログラムの状態が “状態 1” にあったとする．このとき，“イベント 1” が発生するとこのプログラムは “動作 1” を行って “状態 1” に留まる．もしプログラムが “状態 1” にあり，“イベント 2” が発生するとプログラムは “動作 2” を行って “状態 2” に遷移する．すなわち，プログラムが “状態 2” にあるということは，以前にプログラムは “状態 1” にあり，そのとき “イベント 2” が発生した，という記憶を持つことになる．

7.1.1　ネットワークプログラミングと有限状態機械

ネットワークを介して動作するクライアントプログラムやサーバプログラムは有限状態機械で表せることが多い．図 7.1 の状態遷移図に基づくプログラムの例を図 7.2 に示す．イベント発生によって状態が遷移して動作し続けることを想定しているので，全体が無限ループになっている (01-32 行目)．ループの先頭ではイベント発生を待つ (02 行目)．イベント待ちとは，たとえばメッセージ受信待ちである．イベントが発生すると，`switch` 文

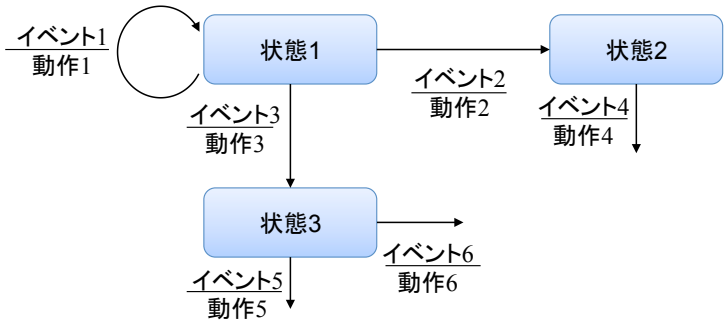

図 7.1: 有限状態機械の例

によりイベントごとの処理を行い，必要であれば状態を更新する．たとえば，状態 1 でイベント 1 が発生した場合 (06 行目) は動作 1() という関数を呼び出す (07 行目)．この場合，状態は変化しない．また，状態 1 でイベント 2 が発生した場合 (09 行目) は動作 2() という関数を呼び出し (10 行目)，状態を更新する (11 行目)．

以上のように，状態遷移図を書くことができればあとは状態遷移図にしたがってプログラミングを進めることができるので，プログラミングも楽であり，バグが入り込む余地も小さくなる．

図 7.1 の状態遷移図に基づくもう 1 つのプログラムの例を図 7.3 に示す．まず動作 1〜6 に対応する関数のプロトタイプ宣言をしている (01-02 行目)．次に構造体 `sstruct proctable` を宣言している (04-08 行目)．この構造体の配列は，どの状態でどのイベントが発生したらどの関数を実行するかを定義するためのものである．そして 09-16 行目で配列の各要素を初期化している．`main` 関数の中では 26 行目から 38 行目が無限ループになっている．27 行目では `wait_event()` という関数の返り値として発生したイベントの種類が変数 `event` に代入される．31 行目から 34 行目の `for` ループの中で，`ptab[]` の中から現在の状態 (変数 `status`) と発生したイベント (変数 `event`) が一致する要素を見つけ (30 行目)，31 行目で登録されている関数を呼び出している．

7.1.2 状態遷移図の例：DHCP の動作

状態遷移の例として DHCP (Dynamic Host Configuration Protocol) を取り上げる．DHCP はコンピュータがネットワークに接続したときに自動的に IP アドレスを割り当てるプロトコルである．ネットワーク上には DHCP サーバが動作しており，ネットワークに接続するコンピュータには DHCP クライアントが動作している．図 7.4 に DHCP が正常に動作した場合のメッセージシーケンスを示し，図 7.5 に状態遷移図を示す．エラーやタイムアウトが発生した場合を省いているので，メッセージシーケンスも状態遷移図もすっきりしている．なお，DHCP サーバの状態遷移図は，1 つの DHCP クライアントに対する状態遷移を示していることに注意して欲しい．DHCP サーバは同時に複数の DHCP クライアントの要求を受け付けることがある．そのようなとき，DHCP サーバは DHCP クライアントごとに状態遷移を管理する必要がある．

DHCP クライアント，サーバとも動作開始時には “初期状態” にある．DHCP サーバは直後に “DISCOVER 待ち” 状態に遷移し，DHCPDISCOVER メッセージの受信を待つ．DHCP クライアントは直ちに DHCPDISCOVER メッセージを送信して “OFFER 待ち” 状態に遷移する．DHCP サーバはクライアントから DHCPDISCOVER メッセージを受信すると割り当て可能な IP アドレスを選択し，DHCPOFFER メッセージをクライアントに返信する．そして “REQUEST 待ち” 状態に遷移する．DHCP クライアントはサーバから DHCPOFFER メッセージを受信すると DHCPREQUEST メッセージを送信し，“ACK 待ち” 状態に遷移する．DHCP サーバはクライアントから DHCPREQUEST メッセージを受信すると DHCPACK メッセージを返信し，再び “DISCOVER 待ち” 状態に遷移し，次のアドレス割り当て要求を待つ．DHCP クライアントはサーバから DHCPACK メッセージを受信すると割り当てられた IP アドレスを設定し，これ以降通信が可能となる．

```
01:    for (;;) {
02:      // イベント待ち (e.g., メッセージ受信待ち)
03:      switch (status) {
04:      case 状態1:
05:        switch (event) {
06:        case イベント1:
07:          動作1();
08:          break;       // 状態遷移無し
09:        case イベント2:
10:          動作2();
11:          status = 状態2;
12:        case イベント3:
13:          動作3();
14:          status = 状態3;
15:          . . .
16:        default:       // 想定外のイベントが発生
17:          エラー処理;
18:          break;
19:        }    // end of case 状態1
20:        break;
21:      case 状態2:
22:        switch (event) {
23:        case イベント4:
24:          動作4();
25:          status = . . .;
26:          . . .
27:        }    // end of case 状態2
28:        break;
29:      case 状態3:
30:        以下, 省略
31:      }   // end of case 状態3
32:    }     // end of for
```

図 7.2: 図 7.1 の状態遷移図に基づくプログラム例

```
01: void f_act1(), f_act2(), f_act3();   |39: void f_act1(...)
02: void f_act4(), f_act5(), f_act6();   |40: {
03:                                      |41:   ...;
04: struct proctable {                   |42: }
06:   int status;                        |43:
06:   int event;                         |44: void f_act2(...)
07:  void (*func)(...);                  |45: {
08: } ptab[] = {                         |46:   ...;
09:   {stat1, event1, f_act1},           |47:   status = stat2;
10:   {stat1, event2, f_act2},           |48: }
11:   {stat1, event3, f_act3},           |49:
12:   {stat2, event4, f_act4},           |50: ...
13:   {stat3, event5, f_act5},           |
14:   {stat3, event6, f_act6},           |---------------------------
15:   ...,
16:   {0, 0, NULL}
17: };
18:
19: int status;
20:
21: int main()
22: {
23:   struct proctable *pt;
24:   int event;
25:
26:   for (;;) {
27:     event = wait_event(...);
28:
29:     for (pt = ptab; pt->status; pt++) {
30:       if (pt->status == status && pt->event == event) {
31:         (*pt->func)(...);
32:         break;
33:       } // end of if() in 30
34:     } // end of for() in 29
35:     if (pt->status == 0)
36:       error processing;
37:   } // end of for() in 26
38: } // end of main()
```

図 7.3: 図 7.1 の状態遷移図に基づくプログラム例 2

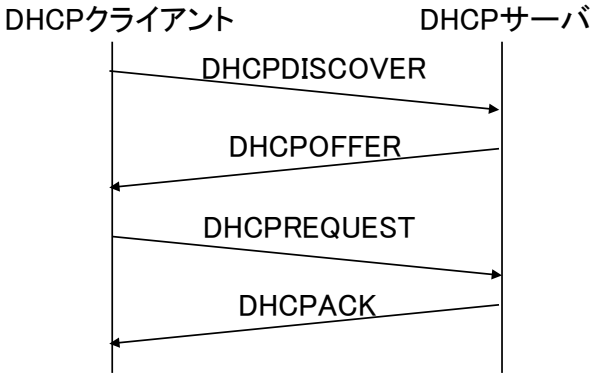

図 7.4: DHCP のメッセージシーケンス (正常な場合)

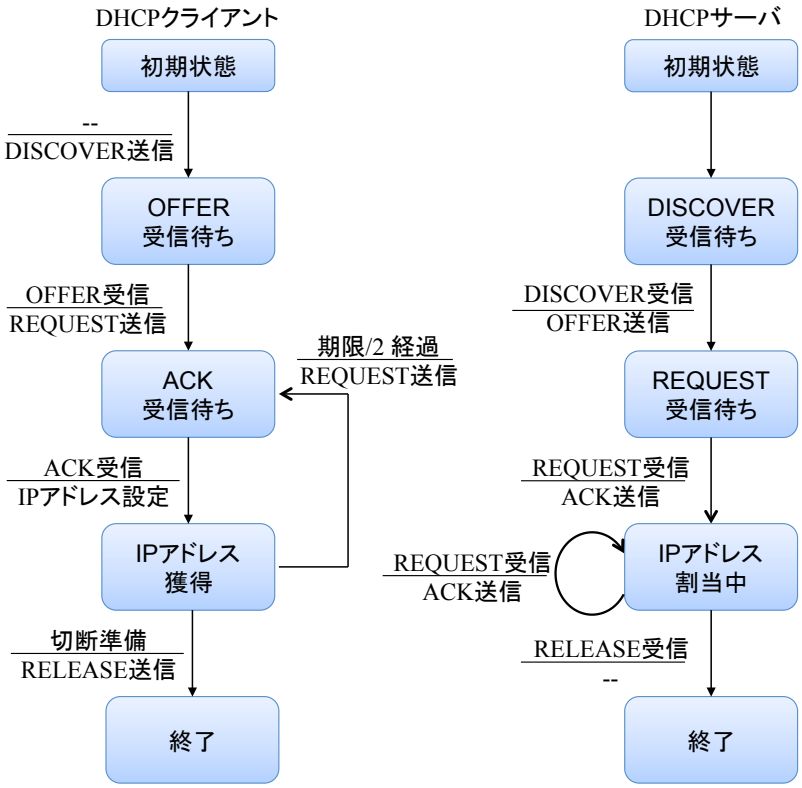

図 7.5: DHCP の状態遷移図 (正常な場合)

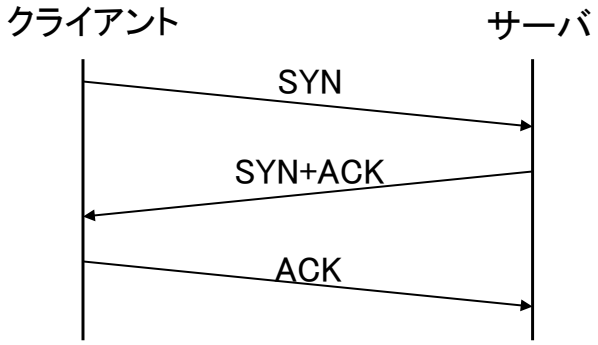

図 7.6: TCP コネクション確立のシーケンス

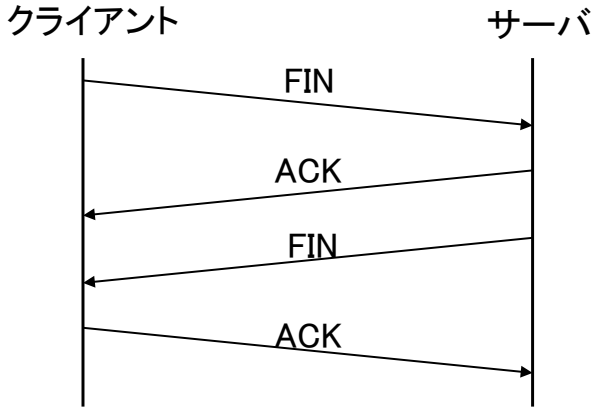

図 7.7: TCP コネクション切断のシーケンス

7.1.3 状態遷移図の例：TCP のコネクション管理

図 7.6 と図 7.7 に TCP コネクション確立のシーケンスおよび TCP コネクション切断のシーケンスを示す．便宜上，TCP コネクションの確立を要求したり切断を要求したりする側をクライアントとし，その相手をサーバと呼ぶことにする．TCP コネクションの確立は 3 つのセグメントの交換によるため，**three-way handshake** と呼ばれる．このような TCP コネクション管理も状態遷移図で表すことができる．図 7.8 に TCP コネクション管理の状態遷移図を示す．この図は TCP の仕様を規定した最初の文書である RFC793 に掲載されているものである．

TCP コネクションの確立について順を追って説明する．クライアント側の状態遷移は以下のようになる．

1. 最初は `CLOSED` 状態である．
2. アプリケーションが `connect()` を実行することにより TCB (Transmission Control Block: TCP コネクションの状態を保持する領域) を確保し，`SYN` セグメントを送信して `SYN SENT` 状態に遷移する．
3. サーバ側が (3) で送信した `SYN+ACK` セグメントを受信することにより `ACK` セグメントを送信し，`ESTABLISHED` 状態に遷移する．

一方，サーバ側の状態遷移は以下のようになる．

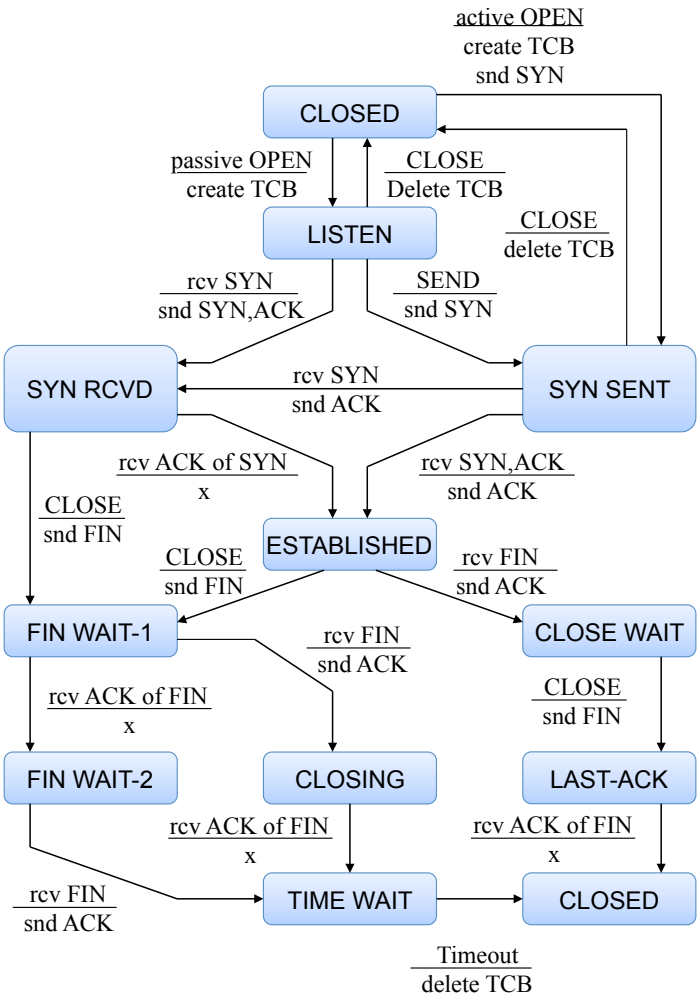

図 7.8: TCP コネクション管理の状態遷移図 (RFC793 から引用)

1. 最初は CLOSED 状態である．
2. アプリケーションが listen() および accept() を実行することにより TCB を確保し，LISTEN 状態に遷移する．
3. クライアント側が (2) で送信した SYN セグメントを受信することにより SYN+ACK セグメントを送信し，SYN RCVD 状態に遷移する．
4. クライアント側が (3) で送信した ACK セグメントを受信し，ESTABLISHED 状態に遷移する．

次に TCP コネクションの切断について順を追って説明する．クライアント側の状態遷移は以下のようになる．

1. アプリケーションが close() を実行することにより FIN セグメントを送信し，ESTABLISHED 状態から FIN WAIT-1 状態に遷移する．
2. サーバ側が (1) で送信した ACK セグメントを受信することにより FIN WAIT-2 状態に遷移する．
3. サーバ側が (2) で送信した FIN セグメントを受信することにより ACK セグメントを送信し，TIME WAIT 状態に遷移する．
4. 決められたタイムアウト時間を待ち，CLOSED 状態に遷移する．

一方，サーバ側の状態遷移は以下のようになる．

1. クライアント側が (1) で送信した FIN セグメントを受信することにより ACK セグメントを送信し，ESTABLISHED 状態から CLOSE WAIT 状態に遷移する．
2. アプリケーションが close() を実行すると FIN セグメントを送信し，LAST-ACK 状態に遷移する．
3. クライアント側が (3) で送信した ACK セグメントを受信することにより CLOSED 状態に遷移する．

7.2 さまざまな受信待ちの方法

7.2.1 ノンブロッキング受信

第 6.4.6 節や第 6.5.5 節では recvfrom() や recv() の使用方法について述べた．これらの関数はパケットが受信されるまでリターンしない．このような受信待ちを**ブロッキング受信**と呼ぶ．パケットが受信されるまで実行がブロック (封鎖) されるからである．これに対し，パケットが受信されていない場合でもすぐに関数 (またはシステムコール) 呼び出しからリターンするような受信待ちを**ノンブロッキング受信**と呼ぶ．

recvfrom() や recv() には int flags という引数があるが，第 6.4.6 節や第 6.5.5 節では flags には 0 を指定しておけばよいと説明した．実は flags の値によっていくつかの

```
while (recv(s, buf, sizeof buf, MSG_DONTWAIT) == 0) {
    // パケット未受信のときの処理
    I/O 待ちなどのない処理;
}
```

図 7.9: ノンブロッキング受信の悪い例

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

int
select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
    struct timeval *timeout);

FD_SET(fd, &fdset);
FD_CLR(fd, &fdset);
FD_ISSET(fd, &fdset);
FD_ZERO(&fdset);
```

図 7.10: select() および関連するマクロの構文

受信方式を指定することができる．ノンブロッキング受信を実行したい場合は flags に MSG_DONTWAIT を指定する．

ノンブロッキング受信を利用することにより，受信パケットを待ちながら他の作業をすることができる．便利であるが，使用する場合には注意が必要である．たとえば図 7.9 に示すようにノンブロッキング受信処理をループしてパケット受信を待つようにすると，無駄に CPU 時間を食い，コンピュータ全体の負荷が上昇してしまう．

7.2.2 select(): 複数の受信待ち

プログラムによっては，複数のソケットにおいて同時にパケット受信を待ちたいことがある．さらにパケット受信とともにファイル (たとえば標準入力) からの入力を同時に待ちたいことがある．このような場合には select() システムコールを利用する．図 7.10 に select() および関連するマクロの構文を示す．

select() は I/O 記述子 (ファイル記述子やソケット記述子) の集合について，入力できる状態か，出力できる状態か，例外事象がペンディング状態か，を判断する．readfds，writefds，exceptfds はそれぞれ入力できる状態かを調べる記述子の集合，出力できる状態かを調べる記述子の集合，例外事象がペンディング状態かを調べる記述子の集合である．nfds は I/O 記述子集合の中で調べる記述子の個数を表す．すなわち，記述子集合の中で 0 番目から nfds−1 番目の記述子が調べる対象となる．select() はすべての記述子

集合の中で入出力などが可能なものの総数を返り値として返す．その際，記述子集合には要求された処理が可能な記述子が含まれる．

記述子集合は整数型オブジェクトのビットフィールドとして表される．記述子集合の操作のためにいくつかマクロが用意されている．FD_ZERO(&fdset) は記述子集合 fdset の各要素を 0 で初期化する．FD_SET(fd, &fdset) は記述子集合 fdset に記述子 fd を含める．FD_CLR(fd, &fdset) は記述子集合 fdset から記述子 fd を削除する．FD_ISSET(fd, &fdset) は記述子 fd が記述子集合 fdset のメンバーの場合は 1 を返し，そうでない場合は 0 を返す．

timeout が NULL ポインタでない場合は待ち時間を表す．時間の扱い方については第 7.3 節で説明する．timeout が NULL ポインタの場合，どれかの記述子が操作可能になるまで select() はブロック (封鎖) 状態となる．ノンブロッキングで select() を使用する場合は 0 を表す timeval 構造体へのポインタを指定する必要がある．

図 7.11 に，標準入力とソケットでのパケット受信を同時に待つ例を示す．03 行目では記述子集合として rdfds を宣言している．14 行目ではまず FD_ZERO() マクロを使用して rdfds を 0 で初期化している．15，16 行目では FD_SET() マクロを使用して標準入力に対応するビットとソケットに対応するビットを記述子集合にセットしている．17 行目で select() を呼び出している．たとえば s の値が 3 の場合は記述子集合の 4 番目までを対象としなければならないので，1 番目の引数は s+1 となっている．この例では入力を待つため，2 番目の引数として rdfds へのポインタを指定し，3 番目および 4 番目の引数は NULL ポインタとなっている．さらにこの例ではタイムアウトを指定しないため，5 番目の引数も NULL ポインタとなっている．標準入力からの入力があったりパケット受信があったりすると select() がリターンする．このとき，入力があった記述子に対応するビットが rdfds にセットされている．そこで 20 行目と 23 行目において FD_ISSET() マクロを使用して入力があったかどうかを調べている．

7.3 時刻や時間の取り扱い

7.3.1 現在時刻の取得

UNIX において現在時刻を得るには date コマンドを用いる．date コマンドは図 7.12 のように現在時刻を表示する．一方，プログラムにおいて現在時刻を得るには gettimeofday() システムコールを用いる．gettimeofday() の構文を図 7.13 に示す．第 2 引数で指定する struct timezone へのポインタは現在は使われていないので，NULL を指定しておけばよい．

gettimeofday() で得られた秒数を時刻表示の文字列に変換するには ctime() 関数を用いる．ctime() の構文を図 7.14 に示す．ctime() の引数としては，gettmeofday() の引数とした struct timeval 型変数のメンバーである tv_sec へのポインタを指定すればよい．このとき，(time_t *) へのキャストを忘れないように．ctime() が返すキャラクタ型ポインタは “Fri Dec 15 21:33:49 2006\n\0” のような 26 文字固定長の文字列を指す．

また，tv_sec の内容を詳細化するために localtime() 関数が用意されている．localtime() の構文を図 7.14 に示す．localtime() は struct tm へのポインタを返す．struct tm に

```
01:    int s;
02:    struct sockaddr_in myskt;
03:    fd_set rdfds;
04:
05:    if ((s = socket(PF_INET, SOCK_DGRAM, 0)) < 0) {
06:        // エラー処理
07:    }
08:
09:    // myskt のメンバーの設定
10:    if (bind(s, &myskt, sizeof myskt) < 0) {
11:        // エラー処理
12:    }
13:
14:    FD_ZERO(&rdfds);
15:    FD_SET(0, &rfds);    // 標準入力を rfds に含める
16:    FD_SET(s, &rfds);    // ソケットを rfds に含める
17:    if (select(s+1, &rfds, NULL, NULL, NULL) < 0) {
18:        // エラー処理
19:    }
20:    if (FD_ISSET(0, &rfds)) {
21:        // 標準入力から入力
22:    }
23:    if (FD_ISSET(s, &rfds)) {
24:        // パケット受信処理
25:    }
```

図 7.11: select() の使用例

```
$ date
Fri Dec 15 21:33:49 JST 2006
$
```

図 7.12: date コマンドの出力例

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

int
gettimeofday(struct timeval *tp, struct timezone *tzp);

struct timeval {
    long tv_sec;    // 1970 年 1 月 1 日からの秒数
    long tv_usec;   // マイクロ秒
};

// struct timezone は現在は使われていない
```

図 7.13: gettimeofday() の構文

は少なくとも表 7.1 に示すメンバーが含まれている．

7.3.2 select() におけるタイムアウトの指定

第 7.2.2 節において select() システムコールの説明をした．select() では第 5 引数で待ち時間のタイムアウトを指定することができる．すなわち，第 5 引数で指定した時間の間に第 2〜4 引数で指定した記述子集合の中で入出力等が可能になったものがない場合には，select() は 0 を返す．このとき，タイムアウトを指定する第 5 引数の型は第 7.3.1 節

```
#include <time.h>

char *
ctime(const time_t *clock);

struct tm *
localtime(const time_t *clock);
```

図 7.14: ctime() と localtime() の構文

表 7.1: struct tm に含まれるメンバー

```
int tm_sec;      // 秒 (0 – 60)
int tm_min;      // 分 (0 – 59)
int tm_hour;     // 時 (0 – 23)
int tm_mday;     // 日 (1 – 31)
int tm_mon;      // 月 (0 – 11)
int tm_year;     // 年 – 1900
int tm_wday;     // 曜日 (日曜日 = 0)
int tm_yday;     // 1 年のうちの通算の日数 (0 – 365)
int tm_isdst;    // サマータイムが有効かどうかのフラグ
char *tm_zone;   // タイムゾーンの名前
long tm_gmtoff;  // 世界標準時 (UTC) からの時差 (秒単位)
```

で説明をした struct timeval である．標準入力からの入力待ちやパケットの受信待ちに関してはタイムアウトを指定して select() を呼び出すことによりノンブロッキングな動作をするプログラムを作成することができる．

7.3.3 タイマによる割り込み

プログラムによっては，ある時点からある一定時間後に何らかの方法でプログラムにその旨を知らせて欲しいことがある．たとえば後述する課題 3 においては，クライアントはサーバから IP アドレスの割り当てを受けるが，IP アドレスには使用期限がついている．クライアントは使用期限の 1/2 が経過したら使用期限延長の処理をしなければならない．

このような場合には setitimer() システムコールを利用する．setitimer() の構文を図 7.15 に示す．オペレーティングシステムはプロセスに 3 種類のインターバルタイマを提供している．1 つ目は ITIMER_REAL タイマであり，実時間で時を刻む．2 つ目は ITIMER_VIRTUAL タイマであり，当該プロセスが動作中のときのみ時を刻む．3 つ目は ITIMER_PROF であるが，これについては省略する．第 1 引数である which によってどのインターバルタイマを使用するかを指定する．第 2 引数である value によってタイムアウトインターバルを指定する．value のメンバーである it_value によってタイムアウトインターバルを指定する．もし it_interval が 0 でない場合，タイムアウト発生後にこの値が it_value に設定される．it_interval が 0 の場合，次回のタイムアウト発生後にはタイマは停止する．it_value に 0 を設定することにより，タイマを停止することができる．第 3 引数である ovalue が NULL ポインタでない場合には，これに以前に設定した value の値が設定される．

ITIMER_REAL がタイムアウトした場合には SIGALRM がプロセスに送信される．また，ITIMER_VIRTUAL がタイムアウトした場合には SIGVALRM がプロセスに送信される．それぞれのシグナルを受信したときに特定の関数を実行したい場合には，第 5.3.12 節や第 5.3.13 節で説明したように，sigaction() システムコールや signal() 関数を使用すればよい．

また，getitimer() システムコールによってインターバルタイマの現在の値を得るこ

```
#include <sys/time.h>
#define ITIMER_REAL    0
#define ITIMER_VIRTUAL 1
#define ITIMER_PROF    2

int
getitimer(int which, struct itimerval *value);

int
setitimer(int which, const struct itimerval *value,
    struct itimerval *ovalue);

struct itimerval {
    struct timeval it_interval;  // タイマインターバル
    struct timeval it_value;     // 現在の値
};
```

図 7.15: getitimer() と setitimer() の構文

とができる．getitimer() の構文を図 7.15 に示す．which で指定したインターバルタイマの値が value に返る．

7.3.4 シグナル受信の待ち方

プログラム中で SIGALRM などのシグナル待ちをする場合，図 7.16 のようにプログラミングしてはならない．この例では，11 行目において SIGALRM が発生したら alrm_func() を実行するように設定している．alrm_func() では外部変数である alrmflag をインクリメントして SIGALRM を受信したことを示すようにしている．一方 main() では 13，14 行目において alrmflag が 0 の間はループして待っている．このようにすると，このプロセスは SIGALRM を待つ間は何も有効な処理をしていないにもかかわらず，CPU 時間を消費する．結果として CPU の負荷が著しく増加してしまう．

シグナル受信を待つには pause() システムコールを使用する．図 7.17 に pause() システムコールの構文を示す．pause() は kill() システムコールや setitimer() システムコールによって送信されるシグナルの受信を待つシステムコールである．シグナルを受け取り，シグナル処理関数の実行が終了すると pause() はリターンする．pause() の返り値は常に-1 である．図 7.18 に正しいシグナルの待ち方の例を示す．

```
01: int alrmflag = 0;
02:
03: void alrm_func()
04: {
05:     alrmflag++;           // SIGALRM の受信を示す
06: }
07:
08: main()
09: {
10:     . . .
11:     signal(SIGALRM, alrm_func);
12:     . . .
13:     while (alrmflag == 0) // SIGALRM の受信待ち
14:         ;                 // alrmflag が 0 の間はループ
15:     . . .
16: }
```

図 7.16: シグナル受信待ちの悪い例

```
#include <unistd.h>

int
pause(void);
```

図 7.17: pause() の構文

```
01: int alrmflag = 0;
02:
03: void alrm_func()
04: {
05:     alrmflag++;          // SIGALRM の受信を示す
06: }
07:
08: main()
09: {
10:     . . .
11:     signal(SIGALRM, alrm_func);
12:     . . .
13:     pause();             // シグナルの受信待ち
14:     if (alrmflag > 0) {  // SIGALRM の受信かの確認
15:         alrmflag = 0;
16:         . . .            // SIGALRM 受信時の処理
17:     }
18: }
```

図 7.18: 正しいシグナル受信待ちの例

7.3.5　シグナル受信とシステムコール，ライブラリ関数

シグナルの送信は実行中のプロセスとは非同期で行われるため，プロセスはいつシグナルを受信するかわからない．通常のプログラムを実行しているときもあれば，システムコールやライブラリ関数を実行中のときもある．pause() システムコールのように明示的にシグナル受信待ちの場合もあれば，read()，recvfrom() などで入力やパケット受信を待っているときにシグナルを受信する可能性もある．いくつかのシステムコールは実行中にシグナルを受信すると実行途中で処理を中断してリターンしてしまうが，signal() システムコールでシグナル処理関数を指定した場合は自動的に元のシステムコールが再実行されるようになっている．

また，シグナル処理関数内で使用するシステムコールやライブラリ関数にも注意が必要である．たとえば，strcpy() 関数内でシグナルを受信してシグナル処理関数の実行が始まり，その中でも strcpy() を呼び出しているとする．すると，strcpy() の実行は途中で中断され，シグナル処理関数内で再度 strcpy() が呼び出されることになる．このとき，strcpy() が再入可能 (reentrant) になっていないと実行結果がおかしくなってしまう．man signal または man sigaction を参照するとシグナル処理関数で使用できるシステムコールやライブラリ関数が列挙されている．しかし，シグナル処理関数においてはなるべく複雑な処理をせず，フラグを設定する程度にすべきである．

7.4　デーモンの作成方法

コンピュータが動作している間システムに常駐し，さまざまな動作を行うプロセスを**デーモンプロセス**と呼ぶことがある．たとえば，/sbin/init プロセスはシステム全体の動作やすべてのプロセスを管理する重要なデーモンプロセスである．ネットワーク関連のサービスを提供するサーバプロセスもデーモンプロセスの一種である．たとえばウェブサーバである httpd，セキュアな遠隔ログインのためのサーバである sshd などさまざまなものがある．

7.4.1　デーモンプロセスの動作

図 7.19 にデーモンプロセスを ps コマンドで表示した例を示す．2 行目には init デーモンが表示されている．4 行目と 5 行目には sshd と httpd が表示されており，これらは両方とも init から起動されていることがわかる．7 行目にも httpd が表示されているが，これは 5 行目の httpd から起動されていることがわかる．同様に，9 行目の sshd は 4 行目の sshd から起動され，さらに 9 行目の sshd から 10 行目の sshd が起動されていることがわかる．

前章の図 6.9 や図 6.19 に UDP 通信や TCP 通信の例を示した．こらの例においては，サーバはクライアントからの要求を待ち，要求を受信するとクライアントへのサービス提供を行い，それが終わるとまた別の要求待ちをするようになっていた．上記の httpd などの実際のデーモンプログラムをこのように作成すると，1 つのクライアントへのサービスが終了するまで別のクライアントからの要求に対応できなくなってしまう．

```
01: USER     PID  PPID  PGID   SID JOBC STAT TT     TIME COMMAND
02: root       1     0     1     1    0 ILs  ??  0:00.04 /sbin/init
03: ...
04: root     573     1   573   573    0 Is   ??  0:00.09 /usr/sbin/sshd
05: root     636     1   636   636    0 Ss   ??  1:45.25 /usr/local/sbin/httpd
06: ...
07: nobody 21594   636   636   636    0 S    ??  0:04.54 /usr/local/sbin/httpd
08: ...
09: root   45792   573 45792 45792    0 Is   ??  0:00.02 sshd: tera
10: tera   45795 45792 45792 45792    0 S    ??  0:00.02 sshd: tera@ttyp0
11: ...
```

図 7.19: デーモンプロセスの例

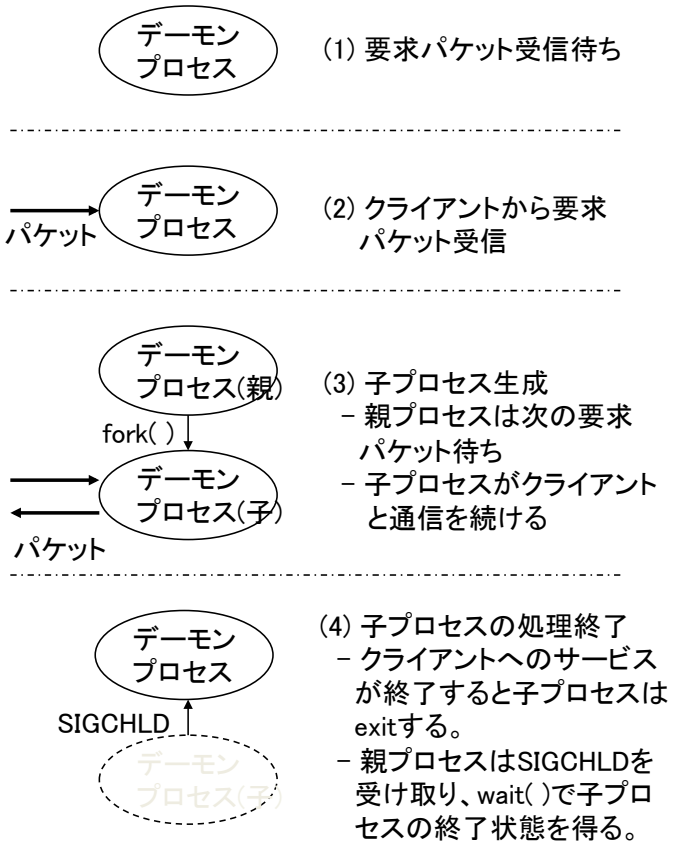

図 7.20: デーモンプロセスの動作

実際にはデーモンプロセスは図 7.20 に示すような動作をする．通常，デーモンプロセスはクライアントからのサービス要求パケットを待ち (図 7.20-(1))，やがてサービス要求パケットを受信する (図 7.20-(2))．するとデーモンプロセスは `fork()` を実行し，子プロセスを生成する (図 7.20-(3))．その後，クライアントへのサービス提供は子プロセスが行い，親プロセスは新たなサービス要求を待つ．クライアントへのサービス提供が終了すると子プロセスは `exit()` を実行して終了する (図 7.20-(4))．すると `SIGCHLD` が親プロセスに送られるので，親プロセスはこれをキャッチし `wait()` を実行して子プロセスの終了状態を得る．

再度，図 7.19 を見ると，上記の動作が確認できる．PID が 636 の `httpd` が親プロセスであり，クライアントからの TCP コネクション接続要求を待っている．クライアントから TCP コネクション接続要求を受信すると `fork()` を実行し，子プロセスを生成する．この図では PID が 21594 の `httpd` である．`sshd` の場合，PID が 537 のものが親プロセスであり，このプロセスから PID が 45792 の子プロセスが生成され，さらにこの子プロセスから PID が 45795 の子プロセスが生成されていることが分かる．

7.4.2 デーモンプロセスが持つべき性質

プロセスは `fork()` システムコールにより親プロセスのコピーとして生成される．その際，さまざまな性質を親プロセスから受け継ぐ．子プロセスのプロセスグループ ID やセッション ID，制御端末なども親プロセスと同一のものになり，親プロセスのカレントディレクトリが子プロセスのカレントディレクトリとなる．親プロセスがオープンしているファイル記述子やソケット記述子は子プロセスに引き継がれ，親プロセスが設定したシグナル処理関数も子プロセスに引き継がれる．子プロセスが `execve()` を実行するとシグナル受信時の動作はデフォルトのものにリセットされるが，`execve()` の実行前に無視すると設定したシグナルについてはそのままになる．

たとえばファイルをオープンしているプロセスが `fork()` を実行すると子プロセスもそのファイルを指すファイル記述子を持つことになる．すると親プロセスと子プロセスのファイル記述子は file table の同一のエントリを指すことになる[1]．オープンしているファイルのどの部分を read/write しているかというオフセット情報は file table のエントリに含まれるため，たとえば子プロセスが `lseek()` を実行すると，親プロセスが行う read/write の位置も変更されてしまう．

また，標準入力からの入力待ちになったり，標準出力や標準エラー出力への出力がエラーになっても困る．端末から送られるシグナルはその端末を制御端末としているすべてのプロセスに送られるため，デーモンとして動作するときにはこのようなシグナルは受信したくない．プロセスはカレントディレクトリを持つ．このとき，そのディレクトリはオープンされている状態にあるので，そのディレクトリを含むファイルシステムを unmount することはできない[2]．

以上のように，子プロセスはさまざまな “しがらみ” を引きずることになり，そのままではデーモンプロセスとして動作するのに不適当である．そこでデーモンとして動作す

[1] file table については第 5.4.1 節を参照．

[2] `df` コマンドを実行すると，ルート (“/”) ファイルシステムに `user` や `var` などの複数のファイルシステムがマウントされていることがわかる．

```
#include <stdlib.h>

int
daemon(int nochdir, int noclose);
```

図 7.21: daemon() の構文

```
#include <stdlib.h>

main()
{
        printf("pid: %d\n", getpid());
        daemon(0, 0);
        pause();
}
```

図 7.22: daemon() のプログラム例

るプロセスは制御端末を切り離し，標準入力，標準出力，標準エラー出力を/dev/null という “無効な” ファイルにリダイレクトし，カレントディレクトリはルートにするなど，デーモンとして動作する前にさまざまな処理を行う必要がある．

7.4.3 daemon(): 制御端末からのプロセスの切り離し

前節で述べたような，デーモンが実際の動作に入る前に行うべきさまざな処理を行うのが daemon() 関数である．daemon() 関数の仕様を図 7.21 に示す．第 1 引数の nochdir が 0 の場合は，このプロセスのカレントディレクトリはルートになる．また第 2 引数の noclose が 0 の場合は，標準入力，標準出力，標準エラー出力は/dev/null にリダイレクトされる．

この関数を呼び出すことにより，プロセスは制御端末から切り離され，バックグラウンドで実行するようになる．daemon() は fork() を実行して親プロセスが終了することにより，子プロセスがバックグラウンドプロセスとして動作するようになる．その際，前節で述べたようなさまざまな処理を行う．

daemon() の効果を見るため，図 7.22 に示すプログラムを実行した結果を図 7.23 に示す．./a.out が daemon() を実行する前のプロセス ID は 35782 であるが，実行後のプロセス ID は 35783 となっており，fork() が実行されていることがわかる．親プロセスが exit() することにより，子プロセスの親プロセスは init に設定される (PPDI = 1)．さらにこの子プロセスは制御端末を切り離し (TT = ??)，自身がセッションリーダとなっていることが分かる (STAT = Ss)．

```
$ ./a.out
pid: 35782
$ ps xj
USER   PID  PPID  PGID   SID JOBC STAT  TT    TIME COMMAND
tera 35783     1 35783 35783    0 Ss    ?? 0:00.00 ./a.out
tera 31211 31210 31211 31211    0 Ss    p3 0:00.03 -tcsh
tera 35784 31211 35784 31211    1 R+    p3 0:00.00 ps xj
$
```

図 7.23: 図 7.22 の実行例

7.5 課題4：擬似DHCPサーバとクライアントの作成

以下に示すような仕様の擬似DHCPクライアント (mydhcpc) および擬似DHCPサーバ (mydhcpd) を作成しなさい．ただし，まずクライアント側，サーバ側ともきちんとした状態遷移図を書き，これに基づいてプログラミングしなさい．また，DHCPサーバは同時に複数のDHCPクライアントの要求を受け付けられるようにしなさい．

7.5.1 擬似DHCPの仕様

本課題で使用するプロトコルを「擬似DHCP」(擬似Dynamic Host Configuration Protocol) と呼ぶこととする．パケットフォーマット，サーバやクライアントの動作を以下に記す．

パケットフォーマット

全体のパケットフォーマットを図7.24に示す．図に示すように，擬似DHCPはUDPを使用する．IPアドレスやポート番号は以下のとおりとする．

- クライアント → サーバ
  - source port: OSが自動的に割り当てる
  - destination port: 51230 (変更してもよい)
  - source IP: OSが自動的に設定する
  - destination IP: サーバのIPアドレス
- サーバ → クライアント
  - source port: 51230 (変更してもよい)
  - destination port: クライアントが使用しているポート番号 (クライアントから受信したパケットで判断)
  - source IP: OSが自動的に設定する
  - destination IP: クライアントのIPアドレス (クライアントから受信したパケットで判断)

擬似DHCPメッセージの詳細なフォーマットを図7.25に示す．Typeフィールドは擬似DHCPメッセージの種類を表す．メッセージの種類は以下のとおりである．メッセージ名の次に記した数字は各メッセージタイプの識別子であり，Typeフィールドに指定される．

- DISCOVER (1)：クライアント → サーバ
  クライアントが稼働中のサーバを検索するためのメッセージ．(注：本来のDHCPはリンク層のマルチキャストによってDHCPDISCOVERメッセージを送信するが，今回は練習問題のためサーバのIPアドレスを明示的に指定している．) Type以外のフィールドは使用されない (0を設定する)．

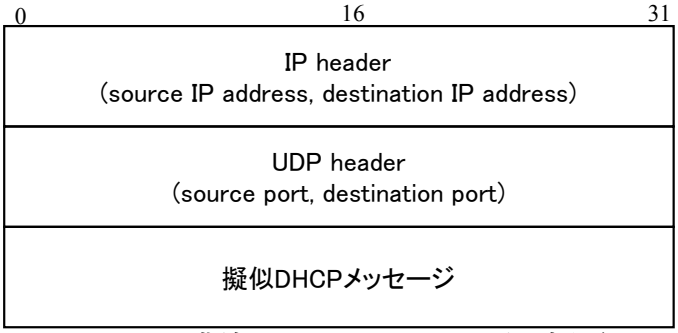

図 7.24: 擬似 DHCP メッセージの概要

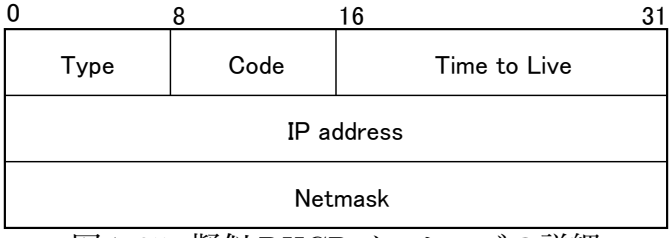

図 7.25: 擬似 DHCP メッセージの詳細

- OFFER (2)：サーバ → クライアント
  サーバからクライアントに割り当て可能な IP アドレスを提示するためのメッセージ．(注：本来の DHCP は，耐故障性のためリンク上に複数の DHCP サーバを設置することができる．そのため，クライアントは複数の DHCPOFFER メッセージを受信する場合がある．今回は練習問題のため，1 台のサーバからのみ OFFER メッセージを受信する．) Code フィールドの値は以下のとおりである．
  - Code = 0: 正常 (割り当て可能な IP アドレスあり)
  - Code = 1: エラー (割り当て可能な IP アドレスなし)

  Time to Live フィールドは IP アドレスの使用期限を秒単位で示す．IP address フィールドと Netmask フィールドは割り当て可能な IP アドレスと netmask を示す．

- REQUEST (3)：クライアント → サーバ
  クライアントからサーバに IP アドレスの割り当てを要求するメッセージ．(注：本来の DHCP では，クライアントが複数のサーバから DHCPOFFER メッセージを受信した場合，何らかの方法によって 1 つのサーバを選択して DHCPREQUEST メッセージを送信する．) Code フィールドの値は以下のとおりである．
  - Code = 2: 割り当て要求
  - Code = 3: 使用期間延長要求

  Time to Live フィールドには希望する IP アドレス使用期限を設定する．なお，この値は OFFER メッセージの Time to Live フィールドで示された値以下でなければならない．IP address フィールドと Netmask フィールドには使用を希望する IP アドレスとネットマスクを設定する．サーバから許可された使用期限以降も IP アドレ

スを使い続ける場合は，使用期限の 1/2 を過ぎたら再度 REQUEST メッセージを送信して新たな使用期限を得る．

- ACK (4)：サーバ → クライアント
  IP アドレス割り当て要求に対する回答のためにサーバがクライアントに送信するメッセージである．Code フィールドの値は以下のとおりである．

  - Code = 0: 正常 (割り当て OK)
  - Code = 4: エラー (REQUEST メッセージに誤りがあった)

  Code = 0 の場合，Time to Live フィールドは IP アドレスの使用期限を秒単位で示す．IP address フィールドと Netmask フィールドは割り当て可能な IP アドレスと netmask を示す．REQUEST メッセージに含まれる IP アドレスやネットマスクが OFFER メッセージで提示されたものと異なる場合は Code = 4 となる．また，REQUEST メッセージに含まれる Time to Live の値が OFFER で提示されたものより大きい場合も Code = 4 となる．

- RELEASE (5)：クライアント → サーバ
  IP アドレスを解放するためにクライアントからサーバに送信するメッセージ．Code フィールド，Time to Live フィールド，Netmask フィールドは使用されない (0 を設定する)．IP address フィールドには解放する IP アドレスを設定する．クライアントは IP アドレスの使用をやめる場合，必ずしも RELEASE メッセージを送信しなければならないわけではない．

正常な場合のメッセージシーケンス

正常な場合のメッセージシーケンスは以下のとおりである (図 7.4 参照)．なお，“C” はクライアントを表し，“S” はサーバを表す．

1. C → S: DISCOVER メッセージを送信．
2. C ← S: OFFER メッセージを送信．割り当て可能な IP アドレス，ネットマスク，使用期限を含む．
3. C → S: REQUEST メッセージを送信．割り当て希望の IP アドレスおよび希望する使用期限を含む．
4. C ← S: ACK メッセージを送信．REQUEST メッセージが受諾されたかどうかを示す．

7.5.2 プログラムの仕様

クライアントプログラムの仕様

クライアントプログラムのファイル名は “mydhcpc” とする．以下に示すように，起動時の引数としてサーバのホスト名を指定する．

```
$ ./mydhcpc server-IP-address
```

クライアントプログラムは以下のように動作するものとする．

- 起動したら，サーバに IP アドレスの割り当てを要求する．
- メッセージを送信したときは，その内容を表示する．
- メッセージを受信したときは，その内容を表示する．
- 状態遷移が発生したときは，どの状態からどの状態に遷移したかを表示する．
- IP アドレスとネットマスクを取得したら，その値と使用期限を表示する．
- 使用期限の 1/2 が経過したら，再度 REQUEST メッセージを送信して使用期限を延長する．
- SIGHUP) を受信したら，RELEASE メッセージを送信して実行を終了する．SIGHUP をクライアントプロセスに送信するには，別のウィンドウでクライアントプロセスの pid を調べ，“$ kill -HUP <pid>” を実行すればよい．
- メッセージ受信待ちのタイムアウトは 10 秒とする．タイムアウトしたらその旨を表示し，再送する．再送もタイムアウトしたら，実行を終了する．
- プロトコルと整合しないメッセージを受信したら，その旨を表示し実行を終了する．

サーバプログラムの仕様

サーバプログラムのファイル名は “mydhcpd” とする．デーモンプロセスにする必要はない．以下に示すように，起動時には割り当て可能な IP アドレスのリストを記した設定ファイルを引数として取る．

```
$ ./mydhcps config-file
```

設定ファイルのフォーマットは以下に示すように，まず 1 行目に使用期限の秒数を整数で表す．2 行目以降は，1 行ごとに割り当て可能な IP アドレスとネットマスクの組をスペース文字 (スペースやタブ) で区切って並べる．

```
使用期限の秒数 (整数)
IP-address-1  netmask-1
IP-address-2  netmask-2
IP-address-3  netmask-3
. . .
```

サーバプログラムは以下のように動作するものとする．

- 起動したら設定ファイルを読み込み，使用期限および割り当て可能なIPアドレスとネットマスクの組を準備する．
- IPアドレスとネットマスクの組を割り当てる際は，最も長く使われていないものから割り当てる．たとえば，IPアドレスとネットマスクの組をリング状のリストで管理し，割り当てるときはリストの先頭のものを使い，回収したときはリストの最後に挿入すれば実現できる．
- メッセージを送信したときは，その内容を表示する．
- メッセージを受信したときは，その内容を表示する．
- 状態遷移が発生したときは，どのクライアントがどの状態からどの状態に遷移したかを表示する．
- IPアドレスとネットマスクの組を割り当てた際は，クライアントのIPアドレス，割り当てたIPアドレスとネットマスクの値，利用期限を表示する．
- 割り当て済みのIPアドレスについて，使用期限が切れたらその旨を表示し，そのIPアドレスとネットマスクの組を回収する．
- REQUESTメッセージ受信待ちのタイムアウトは10秒とする．タイムアウトしたらその旨を表示し，再送する．再送もタイムアウトしたら，該当するクライアントとの処理を終了する．
- プロトコルと整合しないメッセージを受信したらその旨を表示し，該当するクライアントとの処理を終了する．

複数クライアントのサポート

サーバが同時に複数のクライアントからの要求をサポート可能にするためには，サーバはクライアントごとに状態遷移を管理しなければならないことに注意して欲しい．そのためには，たとえば図7.26のような構造体を定義し，クライアントごとに領域を割り当てる．そしてそれらを双方向リストで管理するとよい．新しいクライアントが現れたとき，クライアント管理のための構造体を割当て，双方向リストに接続する．クライアントが明示的にIPアドレスを返却したりIPアドレスの使用期限が切れたりした場合は，そのクライアントを管理する構造体を双方向リストから削除し，構造体の領域を解放する．クライアントごとのサーバの状態は`status`というメンバで管理する．

また，クライアントごとの使用期限の管理が必要である．たとえば以下のように行うとよい．図7.26に示した構造体に`ttlcounter`というメンバがある．IPアドレスを割り当てた際，`ttlcounter`に使用期限を設定しておく．そしてSIGALRMを利用して1秒ごとに`ttlcounter`の値を減算すればよい．

```
struct client {
    struct client *fp;          /* 双方向リスト用ポインタ */
    struct client *bp;          /* 双方向リスト用ポインタ */
    short status;               /* クライアントの状態 */
    int ttlcounter;             /* IP アドレス使用期限の残り時間 */
    // below: network byte order
    struct in_addr id;          /* クライアントの識別子（IP アドレス）*/
    struct in_addr addr;        /* クライアントに割り当てた IP アドレス */
    struct in_addr netmask;     /* クライアントに割り当てた netmask */
    uint16_t ttl;               /* IP アドレスの使用期限 */
};

struct client client_list;    /* クライアントリストのリストヘッド */
```

図 7.26: クライアント管理のための構造体の例

7.6 課題5：FTPサーバとクライアントの作成

課題4ではUDPを使用するクライアント・サーバモデルのプログラムを作成した．本課題ではTCPを使用するクライアント・サーバモデルのプログラムとして，ファイル転送を取り上げる．本課題のファイル転送プロトコルをmyFTPと呼び，クライアントプログラムを`myftpc`，サーバプログラムを`myftpd`と呼ぶこととする．

7.6.1 myFTPの概要

myftpcは以下のようにして起動される．

```
$ ./myftpc <サーバのホスト名>
```

myftpcは引数で指定されたホスト上で動作するmyfptdとの間にTCPコネクションを確立する．TCPコネクションが確立できると，myftpcはプロンプトとして“myFTP%”を表示してユーザからのコマンド入力を待つ．コマンドが入力されると，必要に応じてコマンドメッセージをサーバに送信し，リプライメッセージの受信を待つ．そして再びプロンプトを表示してユーザからのコマンド入力を待つ．

一方，`myftpd`は以下のように起動される．

```
$ ./myftpd [<カレントディレクトリ>]
```

`myftpd`は引数で指定されたディレクトリを初期のカレントディレクトリとする．引数が省略された場合は，`myftpd`が起動されたディレクトリをカレントディレクトリとする．`myfptd`は`myftpc`からのTCPコネクション要求を待つ．`myftpd`はTCPコネクション要求を要求を受信すると`fork()`し，子プロセスがこれ以降のクライアントからのコマンドメッセージを受け付ける．コマンドメッセージを受信すると該当する処理を行い，リプライメッセージをクライアントに返送する．子プロセスはクライアントからのセッション終了要求を受信すると実行を終了する．一方，親プロセスは別のクライアントからのTCPコネクション要求を待つ．

7.6.2 myftpcのコマンド

上述のように，myfptcは起動後に“myFTP%”というプロンプトを表示してユーザからのコマンド入力を待つ．コマンド一覧を表7.2に示す．

quitコマンド：myftpcの終了

`quit`コマンドはmyftpcを終了するためのコマンドである．引数は無い．myftpcはmyftpdとのTCPコネクションをクローズし，実行を終了する．

表 7.2: myftpc のコマンド一覧

コマンド	引数	機能
quit	–	myftpc の終了
pwd	–	サーバでのカレントディレクトリの表示
cd	パス名	サーバでのカレントディレクトリの移動
dir	[パス名]	サーバに存在するファイル情報の取得
lpwd	–	クライアントでのカレントディレクトリの表示
lcd	パス名	クライアントでのカレントディレクトリの移動
ldir	[パス名]	クライアントに存在するファイル情報の取得
get	パス名 1 [パス名 2]	サーバからクライアントへのファイル転送
put	パス名 1 [パス名 2]	クライアントからサーバへのファイル転送
help	–	ヘルプメッセージの表示

pwd コマンド: サーバでのカレントディレクトリの表示

pwd コマンドはサーバでのカレントディレクトリを表示するためのコマンドである．引数は無い．myftpc は PWD コマンドメッセージを myftpd に送信し，リプライメッセージを受信する．

cd コマンド: サーバでのカレントディレクトリの移動

cd コマンドはサーバでのカレントディレクトリを移動するためのコマンドである．引数として移動先のディレクトリのパス名を指定する．myftpc は CD コマンドメッセージを myftpd に送信し，リプライメッセージを受信する．

dir コマンド: サーバでのファイル情報の取得

dir コマンドはサーバに存在するファイル情報を取得するためのコマンドである．引数が存在する場合は，これをパス名と解釈する．パス名はファイル名でもよいし，ディレクトリ名でもよい．myftpc は dir コマンドメッセージを myftpd に送信し，リプライメッセージを受信する．

lpwd コマンド: クライアントでのカレントディレクトリの表示

lpwd コマンドはクライアントでのカレントディレクトリを表示するためのコマンドである．引数は無い．

lcd コマンド: クライアントでのカレントディレクトリの移動

lcd コマンドはクライアントでのカレントディレクトリを移動するためのコマンドである．引数として移動先のディレクトリのパス名を指定する．

ldir コマンド: クライアントでのファイル情報の取得

`ldir` コマンドはクライアントに存在するファイル情報を取得するためのコマンドである．引数が存在する場合は，これをパス名と解釈する．パス名はファイル名でもよいし，ディレクトリ名でもよい．

get コマンド: サーバからクライアントへのファイル転送

`get` コマンドはサーバからクライアントへファイルを転送するためのコマンドである．“パス名 1” で指定されるサーバ上のファイルの内容を，クライアント上の “パス名 2” というファイルに格納する．“パス名 2” が省略された場合はクライアント上のファイル名として “パス名 1” が使用される．

myftpc は `RETR` コマンドメッセージを myftpd に送信し，`OK` リプライメッセージ (`code = 0x01`) を受信する．その後，myftpc は myfptd からデータメッセージを受信し，受信したデータをファイルに格納する．

put コマンド: クライアントからサーバへのファイル転送

`put` コマンドはクライアントからサーバへファイルを転送するためのコマンドである．“パス名 1” で指定されるクライアント上のファイルの内容を，サーバ上の “パス名 2” というファイルに格納する．“パス名 2” が省略された場合はサーバ上のファイル名として “パス名 1” が使用される．

myftpc は `STOR` コマンドメッセージを myftpd に送信し，`OK` リプライメッセージ (`code = 0x02`) を受信する．その後，myftpc は myftpd にデータメッセージを送信し，これを受信した myfptd は受信したデータをファイルに格納する．

7.6.3 myFTP のパケットフォーマット

全体のパケットフォーマットを図 7.27 に示す．myFTP は TCP 上で動作する．IP アドレスやポート番号は以下のとおりとする．

- クライアント → サーバ
  - source port: OS が自動的に設定する
  - destination port: 50021 (変更してもよい)
  - source IP: OS が自動的に設定する
  - destination IP: サーバの IP アドレス (myftpc の起動時に指定する)
- サーバ → クライアント
  - source port: 50021 (変更してもよい)
  - destination port: クライアントが使用しているポート番号

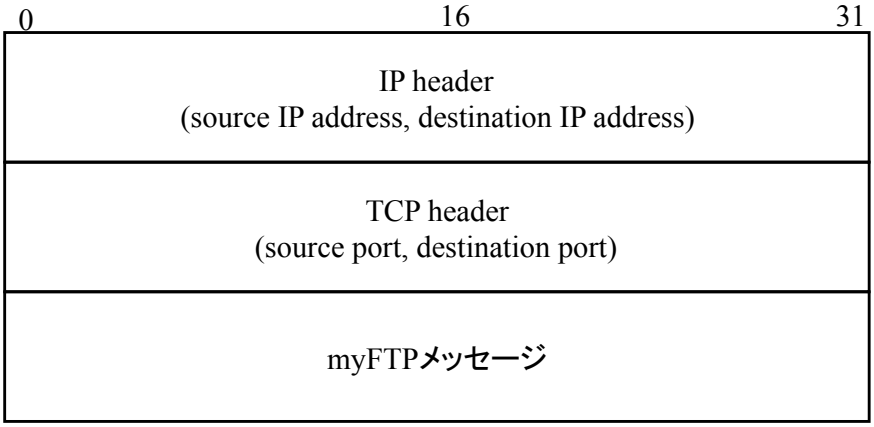

図 7.27: myFTP メッセージの概要

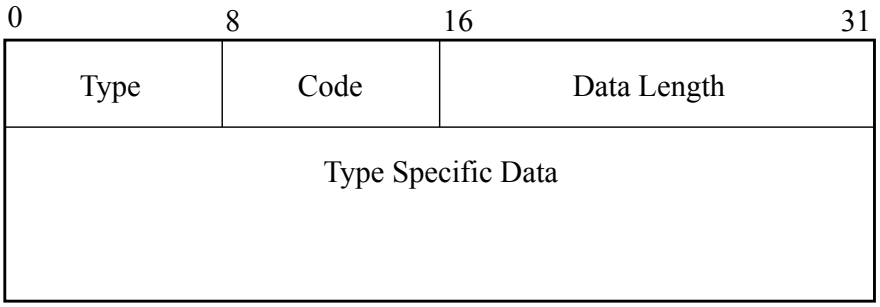

図 7.28: myFTP メッセージの詳細

- source IP: OS が自動的に設定する
- destination IP: クライアントの IP アドレス

myFTP メッセージの詳細なフォーマットを図 7.28 に示す．`Type` フィールドは myFTP メッセージの種類を表す．さらに `Code` フィールドはいくつかのメッセージタイプに関してさらに詳しい情報を表す．表 7.3 に myFTP メッセージの一覧を示す．myFTP メッセージタイプはコマンドメッセージ，リプライメッセージおよびデータメッセージに分けられる．コマンドメッセージはクライアントからサーバに送信され，リプライメッセージはコマンドメッセージの応答としてサーバからクライアントに送信される．データメッセージはサーバからクライアント，またはクライアントからサーバに送信される．表 7.3 の上段はコマンドメッセージ，中段はリプライメッセージ，下段はデータメッセージである．

QUIT コマンド

`QUIT` コマンドは myFTP を終了するコマンドである．クライアントからサーバに送信されるコマンドメッセージにおいて，`Type` フィールドの値は `0x01` (`QUIT`) である．`Data` フィールドは存在せず，`Data Length` フィールドの値は `0` である．

表 7.3: myFTP メッセージ一覧

Type	Code	機能	説明
0x01	–	QUIT	myFTP セッションの終了通知
0x02	–	PWD	サーバでのカレントディレクトリを取得
0x03	–	CWD	サーバでのディレクトリの移動
0x04	–	LIST	サーバでのディレクトリ情報を取得
0x05	–	RETR	サーバからのファイル転送
0x06	–	STOR	サーバへのファイル転送
0x10	0x00	OK	コマンド OK
0x10	0x01	OK	コマンド OK．DATA メッセージが続く (サーバ → クライアント)
0x10	0x02	OK	コマンド OK．DATA メッセージが続く (クライアント → サーバ)
0x11	0x01	CMD-ERR	エラー: 構文エラー
0x11	0x02	CMD-ERR	エラー: 未定義コマンド
0x11	0x03	CMD-ERR	エラー: プロトコルエラー
0x12	0x00	FILE-ERR	エラー: ファイル/ディレクトリが存在しない
0x12	0x01	FILE-ERR	エラー: ファイル/ディレクトリのアクセス権限がない
0x13	0x05	UNKWN-ERR	エラー: 未定義のエラー
0x20	0x00	DATA	データ，後続なし
0x20	0x01	DATA	データ，後続あり

通常，サーバにおいてこのコマンドの実行は成功するはずである．サーバがクライアントに送信するリプライメッセージにおいて，Type フィールドの値は 0x10 (OK) であり，Code フィールドの値は 0x00 である．Data フィールドは存在せず，Data Length フィールドの値は 0 である．

PWD (Print Working Directory) コマンド

PWD コマンドは，サーバでのカレントディレクトリのパス名を取得するコマンドである．クライアントからサーバに送信されるコマンドメッセージにおいて，Type フィールドの値は 0x02 (PWD) である．Data フィールドは存在せず，Data Length フィールドの値は 0 である．

通常，サーバにおいてこのコマンドの実行は成功するはずである．サーバがクライアントに送信するリプライメッセージにおいて，Type フィールドの値は 0x10 (OK) であり，Code フィールドの値は 0x00 である．Data フィールドにはカレントディレクトリのパス名が格納される．パス名の最後には改行文字や空文字 ('\0') を含まないものとする．Data Length フィールドはパス名の長さを格納する．

CWD (Current Working Directory) コマンド

CWD コマンドは，サーバでカレントディレクトリを移動するためのコマンドである．クライアントからサーバに送信されるコマンドメッセージにおいて，Type フィールドの値は 0x03 (CWD) である．Data フィールドには移動先のディレクトリのパス名が格納される．パス名の最後には改行文字や空文字 ('\0') を含まないものとする．Data Length フィールドにはパス名の長さが格納される．

サーバにおいてカレントディレクトリの移動が成功した場合，サーバがクライアントに送信するリプライメッセージにおいて，Type フィールドの値は 0x10 (OK) であり，Code フィールドの値は 0x00 である．Data フィールドは存在せず，Data Length フィールドの値は 0 である．

サーバにおいてカレントディレクトリの移動が失敗した場合，リプライメッセージの Type フィールドと Code フィールドにはエラーの原因を表す値が格納される．Data フィールドは存在せず，Data Length フィールドの値は 0 である．

LIST コマンド

LIST コマンドは，サーバでのカレントディレクトリに存在するファイルの情報を取得するコマンドである．具体的には "ls -l" の実行結果のような情報を返すものとする (表示結果は必ずしも "ls -l" と同じでなくてもよい)．クライアントからサーバに送信されるコマンドメッセージにおいて，Type フィールドの値は 0x04 (LIST) である．Data フィールドには対象となるディレクトリのパス名が格納される．パス名の最後には改行文字や空文字 ('\0') を含まないものとする．Data Length フィールドにはパス名の長さが格納される．Data フィールドが省略された場合はカレントディレクトリを対象とする．この場合，Data Length フィールドの値は 0 となる．

サーバにおいてファイル情報の取得に成功した場合，サーバがクライアントに送信するリプライメッセージの Type フィールドの値は 0x10 (OK) であり，Code フィールドは 0x01 である．Data フィールドは存在せず，Data Length フィールドの値は 0 である．このリプライメッセージのあとにデータメッセージが続く．データメッセージの Type フィールドの値は 0x20 である．データメッセージが後続する場合，Code フィールドの値は 0x01 であり，このデータメッセージが最後の場合は 0x00 となる．Data フィールドにはディレクトリ情報の一部あいるは全部が格納される．ディレクトリ情報の最後には改行文字や空文字 ('\0') を含まないものとする．Data Length フィールドには，Data フィールドに格納されたデータのバイト数が格納される．

サーバにおいてファイル情報の取得が失敗した場合，リプライメッセージの Type フィールドと Code フィールドにはエラーの原因を表す値が格納される．Data フィールドは存在せず，Data Length フィールドの値は 0 である．

RETR (Retrieve) コマンド

RETR コマンドはサーバ上のファイルをクライアントマシンに転送するコマンドである．クライアントからサーバに送信されるコマンドメッセージにおいて，Type フィールドの

値は 0x05 (RETR) である．Data フィールドにはサーバ上のファイルのパス名が格納される．パス名の最後には改行文字や空文字 ('\0') を含まないものとする．Data Length フィールドにはパス名の長さが格納される．

サーバにおいて指定されたファイルにアクセス可能な場合，サーバがクライアントに送信するリプライメッセージにおいて，Type フィールドの値は 0x10 (OK) であり，Code フィールドの値は 0x01 となる．Data フィールドは存在せず，Data Length フィールドの値は 0 である．このリプライメッセージのあとにデータメッセージがサーバからクライアントに送信される．データメッセージの Type フィールドの値は 0x20 である．データメッセージが後続する場合，Code フィールドの値は 0x01 であり，このデータメッセージが最後の場合は 0x00 となる．Data フィールドにはファイルのデータの一部あいるは全部が格納される．Data Length フィールドには，Data フィールドに格納されたデータのバイト数が格納される．

サーバにおいて指定されたファイルにアクセスできない場合，リプライメッセージの Type フィールドと Code フィールドにはエラーの原因を表す値が格納される．Data フィールドは存在せず，Data Length フィールドの値は 0 である．

STOR (Store) コマンド

STOR コマンドはクライアントマシン上のファイルをサーバマシンに転送するコマンドである．クライアントがサーバに送信するコマンドメッセージにおける Type フィールドの値は 0x06 (STOR) である．Data フィールドにはサーバ上のファイルのパス名が格納される．パス名の最後には改行文字や空文字 ('\0') を含まないものとする．Data Length フィールドにはパス名の長さが格納される．

サーバにおいて指定されたファイルにアクセス可能な場合，サーバがクライアントに送信するリプライメッセージにおける Type フィールドの値は 0x10 (OK) であり，Code フィールドの値は 0x02 となる．Data フィールドは存在せず，Data Length フィールドの値は 0 である．

クライアントがこのリプライメッセージを受信すると，クライアントはサーバにデータメッセージを使用してファイルのデータを送信する．データメッセージの Type フィールドの値は 0x20 である．データメッセージが後続する場合，Code フィールドの値は 0x01 であり，このデータメッセージが最後の場合は 0x00 となる．Data フィールドにはファイルのデータの一部あいるは全部が格納される．Data Length フィールドには，Data フィールドに格納されたデータのバイト数が格納される．

サーバにおいて指定されたファイルにアクセスできない場合，リプライメッセージの Type フィールドと Code フィールドにはエラーの原因を表す値が格納される．Data フィールドは存在せず，Data Length フィールドの値は 0 である．

練習問題

1. 以下に示すようなプログラムを作り，2 台のホストでこのプログラムを実行してみなさい．

   - 起動後，通信相手の IP アドレスをキーボードから読み込む．その後，以下を繰り返す．
   - UDP のポート 49155 で受信を待つと同時にキーボードからの入力を待つ．
   - キーボードからの入力があったらその文字列を通信相手に送信する．
   - キーボードからの入力が “FIN” だった場合はこの文字列を送信した後，実行を終了する．
   - UDP ポートから受信したら受信データを表示する．
   - UDP ポートから受信したデータが “FIN” だった場合は，実行を終了する．

2. 以下に示すようなサーバを作成しなさい．サーバは複数のクライアントに同時にサービスを提供するものとする．ただし，送信されるデータはすべて文字列とする．

   - UDP のポート 49158 でクライアントからの受信を待ち，受信したデータが “START” の場合はそのクライアントへのサービスを開始する．サービスを開始したら，その旨とクライアントの IP アドレスとポート番号を表示する．
   - サービス中のクライアントからデータを受信したら，それを表示する．
   - サービス中のクライアントから受信したデータが “FIN” の場合，そのクライアントへのサービスを終了し，その旨を表示する．

3. 以下に示すようなサーバを作成しなさい．サーバは複数のクライアントに同時にサービスを提供するものとする．ただし，送信されるデータはすべて文字列とする．

   - TCP のポート 49158 でクライアントからのコネクション要求を待ち，コネクションが確立したら子プロセスを生成してクライアントへのサービスを開始する．サービスを開始したら，その旨とクライアントの IP アドレスとポート番号を表示する．親プロセスは引き続きポート 49152 でクライアントからのコネクション要求を待つ．
   - サービス中のクライアントからデータを受信したら，それを表示する．
   - サービス中のクライアントから受信したデータが “FIN” の場合，そのクライアントへのサービスを終了し，その旨を表示する．そしてプロセスを終了する．