UNIXシステムプログラミング

第11回 クライアント・サーバのための プログラミング(2)

2019年12月20日

情報工学科

寺岡文男

タイマによる割込み

- 一定時間ごとに特定の処理をしたい
  - sigaction( ) または signal( ) でSIGALRM/SIGVTALRMの処理関数を登録
  - setitimer( )により，指定した時間間隔でSIGALRM/SIGVTALRMを発生
  - mainのスレッドでSIGALRM/SIGVTALRMを待つ場合はpause( )を使用
- OSはプロセスに3種類のタイマを提供
  - ITIMER_REAL: 実時間で進むタイマ → SIGALRM
  - ITIMER_VIRTUAL: プロセスが実行状態のときだけ進むタイマ → SIGVTALRM
  - ITIMER_PROF: 省略

setitimer( ): インターバルタイマ

```
#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, 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
};
```

setitimer( ): インターバルタイマ (cont'd)

- setitimer( )の引数
  - int which: 3種類のタイマのうちどれを使うか
  - struct itimerval *value:
    - メンバー it_valueでタイムアウトインターバルを指定
    - メンバーit_valueの内容に0を設定することによりタイマを停止
    - メンバーit_intervalがノンゼロの場合，タイムアウトが発生するごとにこの値がメンバーit_valueに設定される
    - メンバーit_intervalの内容が0の場合は次のタイムアウトによりタイマは停止
  - struct itimerval *ovalue:
    - non nilの場合，以前に設定した値が返される

シグナルの受信待ち (悪い例)

```
01: int alrmflag = 0;
02:
03: void alrm_func()
04: {
05:   alrmflag++;
06: }
07:
08: main()
09: {
10:   . . .
11:   sigaction(SIGARLM, . . .);
12:   . . .
13:   while (alrmflag == 0)
14:         ;
15:   alrmflag = 0;
16:   . . .
17: }
```

SIGALRMを受信したら
フラグをインクリメント

SIGALRMを受信したら
alrm_func( )を実行するように
設定

ビジーウェイト
→ CPU時間を浪費

シグナルの受信待ち (良い例)

```
01: int alrmflag = 0;
02:
03: void alrm_func()
04: {
05:   alrmflag++;
06: }
07:
08: main()
09: {
10:   . . .
11:   sigaction(SIGARLM, . . .);
12:   . . .
13:   pause();                    // シグナル受信待ち
14:   if (alrmflag > 0) {         // SIGALRMか?
15:         alrmflag = 0;         // フラグをリセット
16:         . . .                 // SIGALRM受信時の処理
17:   }
18: }
```

SIGALRMを受信したら
alrm_func( )を実行するように
設定

pause( ): シグナルの受信待ち

```
#include <unistd.h>

int pause(void);
```

- 機能
  - kill( )やsetitimer( )などが送信するシグナルの受信を待つ。
    - CPU時間を消費しない
- 返り値
  - 返り値は常に –1

シグナル受信とシステムコール

- シグナルをいつ受信するかは分からない
  - システムコール実行中，ライブラリ関数実行中，通常の処理中，pause( )で明示的に待機中，etc.
- システムコール実行中 → 途中で処理を打切り
  - int errno にEINTRが設定される
- signal( )でシグナル処理関数を設定，またはsigaction( )でSA_RESTARTフラグを設定
  → システムコールは自動的に再度実行される
  - read( ), write( ), sendto( ), recvfrom( ), etc.

シグナル受信と関数

- シグナル処理関数の中で利用する関数は再入可能(reentrant)でなければならない.
  - sigaction( )のmanページなどを見ると，シグナル処理関数内で使用できる関数が列挙されている．(BSDの場合)
  - Linuxの場合，“man 7 signal” のページを参照

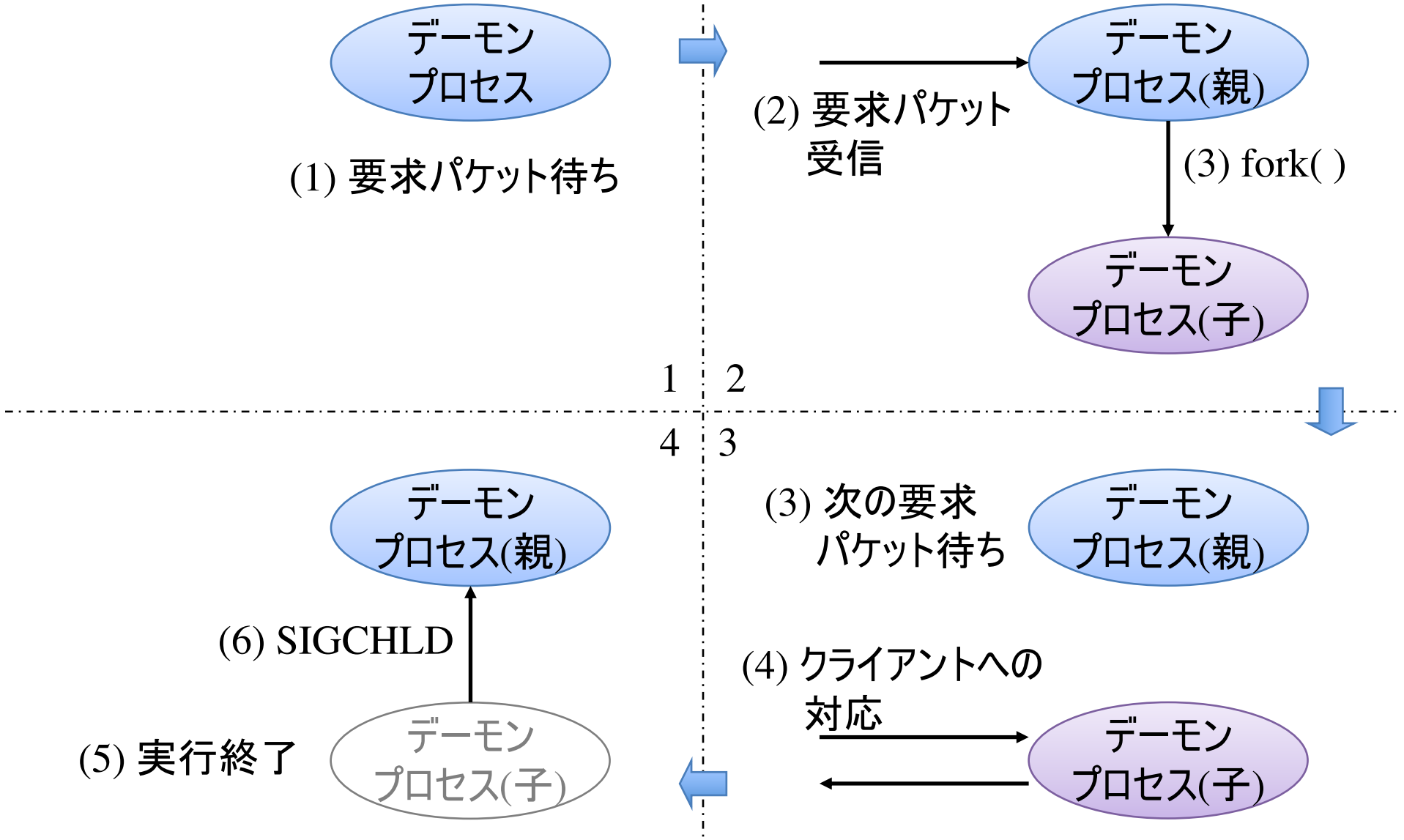
- 基本的にシグナル処理関数での処理はなるべく簡単にする方がよい
  - フラグのセットなど.

デーモンの動作

- デーモン: コンピュータに常駐し，さまざまなサービスを提供するプロセス
  - /sbin/init, メールサーバ, httpサーバ, etc.
- デーモンプロセスの例 (“ps –axj” の実行例)

```
USER      PID  PPID  PGID   SID JOBC STAT TT     TIME COMMAND
root        1     0     1     1    0 ILs  ?? 0:00.04 /sbin/init
. . .
root      573     1   573   573    0 Is   ?? 0:00.09 …/sshd
root      636     1   636   636    0 Ss   ?? 1:45.25 …/httpd
. . .
nobody 21594   636   636   636    0 S    ?? 0:04.54 …/httpd
. . .
root   45792   573 45792 45792    0 Is   ?? 0:00.02 sshd:…
tera   45795 45792 45792 45792    0 S    ?? 0:00.02 sshd:…
. . .
```

デーモンの動作


デーモン
プロセス
(1) 要求パケット待ち
(2) 要求パケット
受信
デーモン
プロセス(親)
(3) fork( )
デーモン
プロセス(子)
1
2
4
3
デーモン
プロセス(親)
(6) SIGCHLD
(5) 実行終了
デーモン
プロセス(子)
(3) 次の要求
パケット待ち
デーモン
プロセス(親)
(4) クライアントへの
対応
デーモン
プロセス(子)

デーモンプロセスが持つべき性質

- プロセスはfork( )で生成 → 親から子に性質が継承
  - プロセスグループID，セッションID，制御端末
  - カレントディレクトリ，オープンしているファイルやソケット
  - シグナル処理関数
- 悪影響の例：オープンしているファイル
  - ファイルをオープンし，fork( ) → fdの指す先を親子で共有
  - 子がread( ) → オフセット変化
  - 親がread( ) → 子のread( )のあとのデータを読み込む
- デーモンとして動作するには，さまざまな “しがらみ” を断ち切る必要あり

daemon( ): 制御端末からの切り離し

```
#include <stdlib.h>

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

- 機能
  - プロセスがデーモンとして動作するためのさまざまな処理を行う
  - 制御端末から切り離され，バックグラウンドへ移行する
- 引数
  - int nochdir: ノンゼロの場合，“/” にcd しない
  - int noclose: ノンゼロの場合，stdin, stdout, stderrを/dev/nullにリダイレクトしない

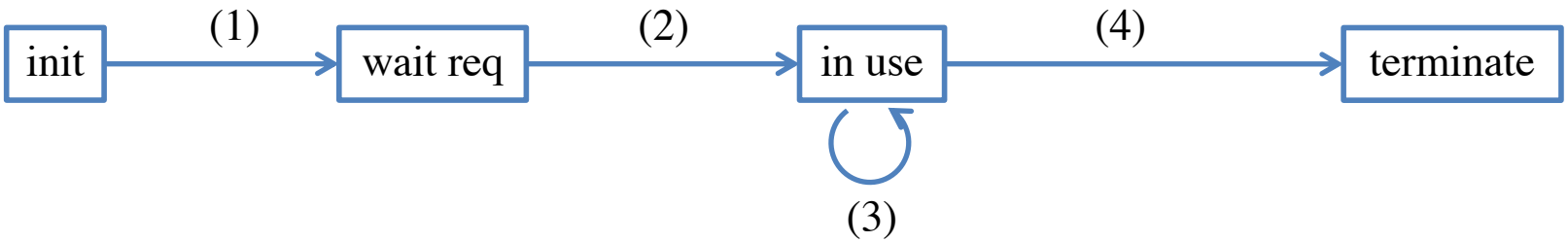
daemon( )の実行例

```
#include <stdlib.h>
main()
{
        printf("pid %d¥n", getpid());
        daemon(0, 0);
        pause();
}
```

```
% ./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 31211 31211    1 R+   p3 0:00.00 ps xj
%
```

課題4：mydhcpの注意事項・ヒント

サーバの状態遷移図 (クライアントごと)



(1)

recv DISCOVER
create client, alloc IP, send OFFER

(2)

recv REQUEST (alloc) [OK]
send ACK [OK]

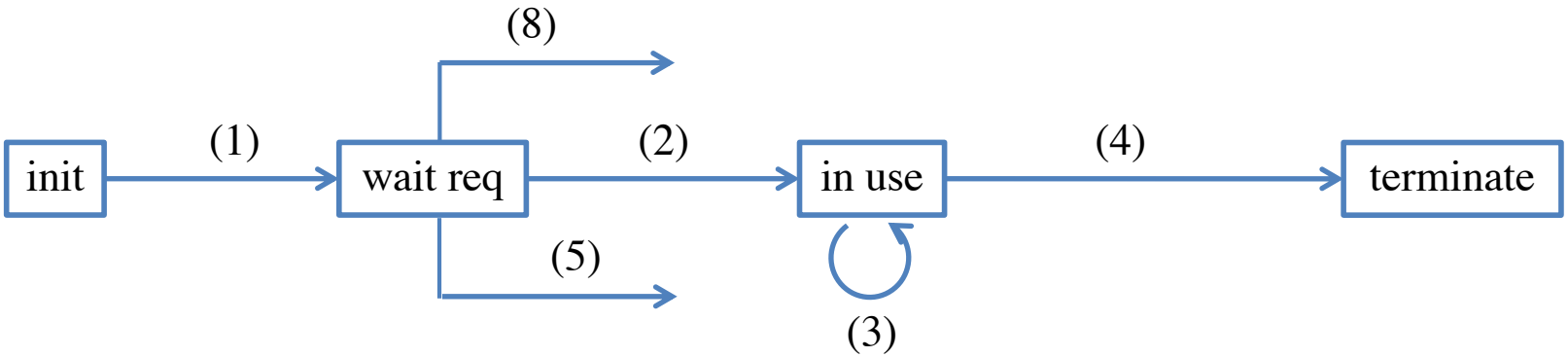
(4)

recv RELEASE [OK] || TTL timeout
recall IP, del client

(3)

recv REQUEST (ext) [OK]
reset TTL, send ACK [OK]

サーバの状態遷移図 (クライアントごと)



(1)

recv DISCOVER
create client, alloc IP, send OFFER

(2)

recv REQUEST (alloc) [OK]
send ACK [OK]

(4)

recv RELEASE [OK] || TTL timeout
recall IP, del client

(5)

recv timeout
??

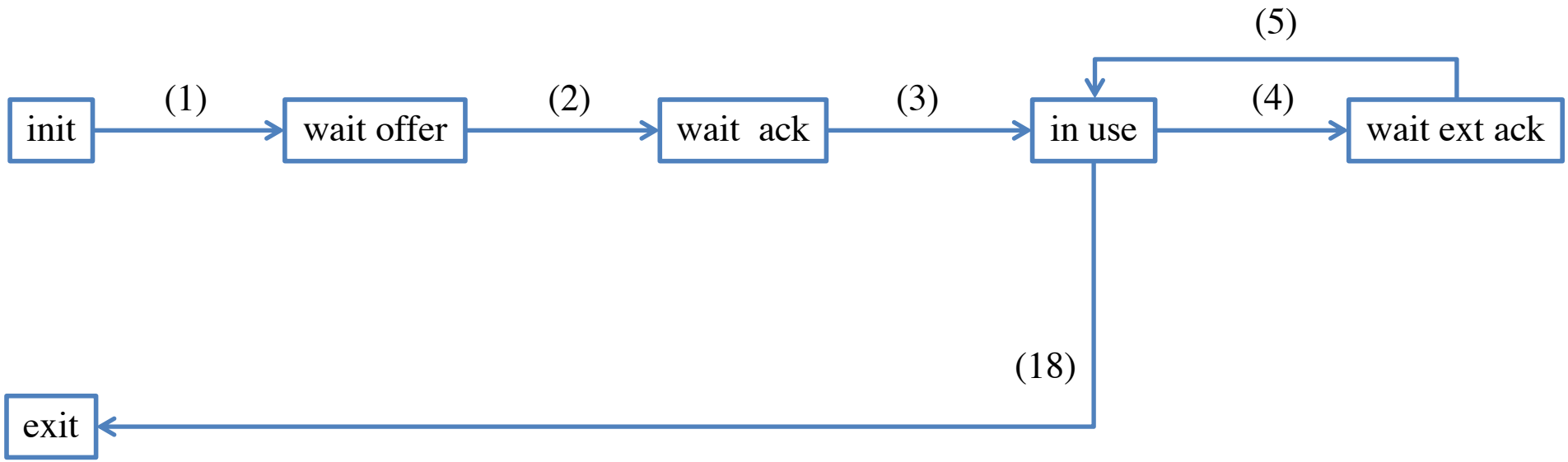
(3)

recv REQUEST (ext) [OK]
reset TTL, send ACK [OK]

(8)

recv REQUEST (alloc) [NG]
??

クライアントの状態遷移図



(1)
--
send DISCOVER

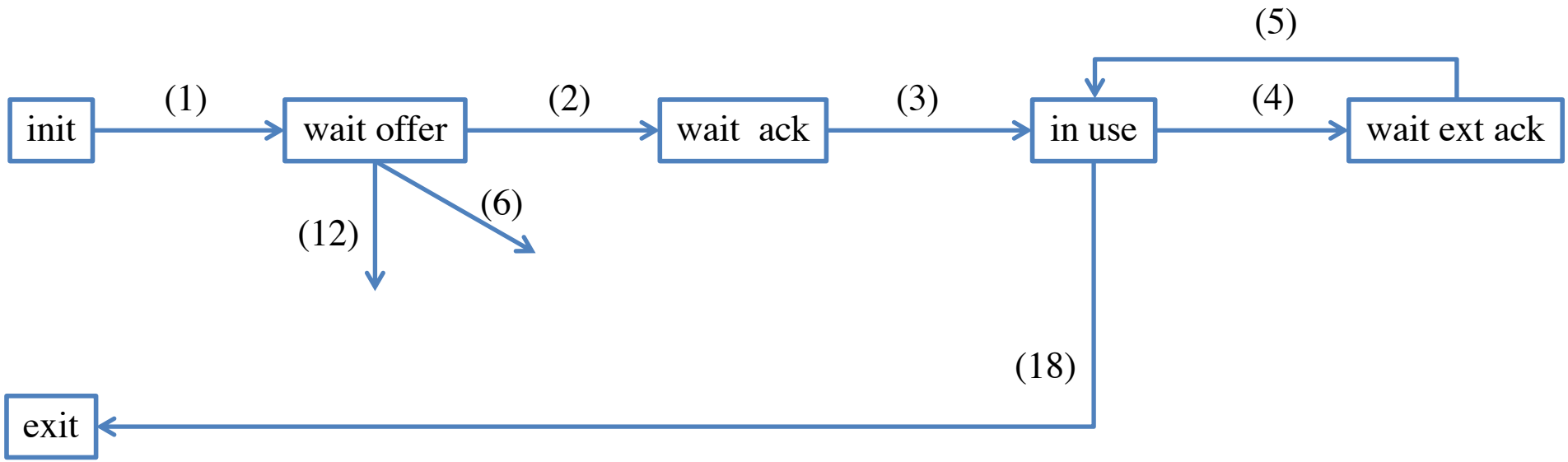
(2)
recv offer [OK]
send REQUEST [alloc]

(3),(5)
recv ACK [OK]
--

(4)
TTL/2 passed
send REQUEST [ext]

(18)
recv SIGHUP
send RELEASE

クライアントの状態遷移図



(1) -- / send DISCOVER

(6) recv timeout / re-send DISCOVER

(2) recv offer [OK] / send REQUEST [alloc]

(3),(5) recv ACK [OK] / --

(18) recv SIGHUP / send RELEASE

(4) TTL/2 passed / send REQUEST [ext]

(12) recv offer [NG] / ??

注意事項

- サーバはクライアントごとにfork()しない
  - fork()すると子プロセス間で受信パケットの奪い合いになる
  - IPアドレスプールは共有資源

- サーバはデーモンにする必要はない
  - デバッグが面倒になるので

- サーバは複数のクライアントをリスト構造などで管理
  - メッセージ受信 → 該当するクライアントを特定
    → クライアントの状態とメッセージタイプで処理を決定

(再掲) FSMをプログラムへ: 例2 (1)

```
void f_act1(...), f_act2(...), f_act3(...), ...;

struct proctable {
  int status;
  int event;
  void (*func)(...);
} ptab[ ] = {
  {stat1, event1, f_act1},
  {stat1, event2, f_act2},
  {stat1, event3, f_act3},
  {stat2, event4, f_act4},
  {stat3, event5, f_act5},
  {stat3, event6, f_act6},
  ...,
  {0, 0, NULL}
};

int status;
```

(再掲) FSMをプログラムへ: 例2 (2)

```
int main( )
{
  struct proctable *pt;
  int event;

  for (;;) {
    event = wait_event(...);
    for (pt = ptab; pt->status; pt++) {
      if (pt->status == status &&
                    pt->event == event) {
        (*pt->func)(...);
        break;
      }
    if (pt->status == 0)
      エラー処理;
  }
}
```

```
void f_act1(...)
{
  ...;
}

void f_act2(...)
{
  ...;
  status = stat2;
}

...
```

ヒント：wait_event( )の処理

- クライアント側
  - アドレス取得前：メッセージ受信を待つ
  - アドレス取得後：SIGALRM or SIGHUPを待つ

- サーバ側
  - メッセージ受信のタイムアウトをチェック
  - 使用期限タイムアウトをチェック
  - メッセージ受信を待つ

ヒント：サーバにおけるクライアント管理

- クライアント管理用リストの例
  - クライアントごとに構造体を割当て，双方向リストで管理
  - タイムアウト管理：SIGALRMを1秒ごとに発生させ，ttlcounterをdecrementする．

```
struct client {
    struct client *fp;          // for client management
    struct client *bp;
    int stat;                   // client state
    int ttlcounter;             // start time
    // below: network byte order
    struct in_addr id;          // client ID (IP address)
    struct in_addr addr;        // allocated IP address
    struct in_addr netmask;     // allocated netmask
    in_port_t port;             // client port number
    uint16_t ttl;               // time to live
};

struct client client_list;      // client list head
```

ヒント：サーバにおける処理

- メッセージ受信
  - 始点IPアドレスと始点ポート番号でクライアントを識別
  - client リストを検索
  - clientが存在しない場合，新しくエントリを割り当てる

- クライアントの状態とメッセージタイプで処理関数決定