[7-4.] ネットワークサービスは必ず fork しよう

管理用のポートだからということで1つしか接続を受け付けないネットワークサービスがある。このときプロセスをforkしていないと，そのポートを占有されて業務妨害に陥ることがある。

● ● ●

複数クライアントの同時接続と DoS 攻撃

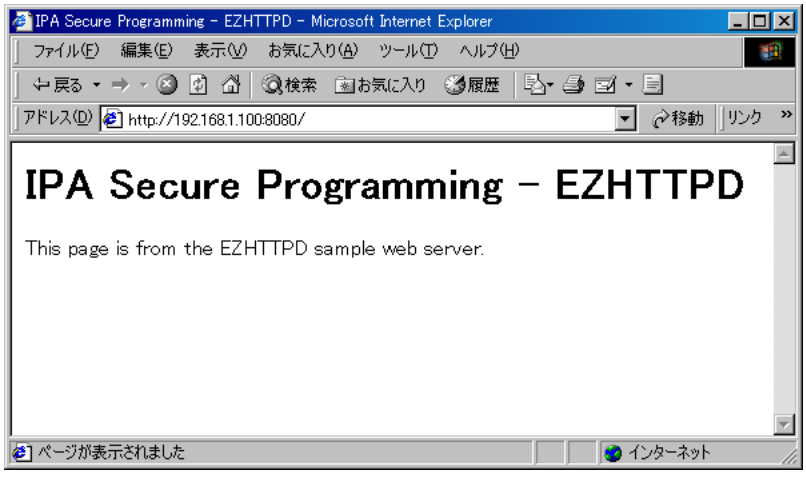
ネットワークサービスの多くはクライアント・サーバモデルに基づき設計されている。インターネット上のネットワークサービスとして有名なHTTP（WWW)やTelnet，FTP，SMTP，POP3もクライアント・サーバモデルに基づき設計されている。これらのサーバプログラムは，複数のクライアント接続を同時に処理できるように設計されている。しかし複数クライアントによる同時接続を処理するように作られていないサーバプログラムはDoS攻撃（サービス妨害攻撃）を受ける可能性がある。

複数クライアントの同時接続を考慮していない簡易 HTTP サーバ

リスト１に複数クライアントの同時接続を考慮していない簡易HTTPサーバのサンプルプログラムを示す。このプログラムは文末の**リスト４**と**リスト５**の関数を使用しているので，合わせて参照していただきたい。

このサンプルプログラムはInternet ExplorerやNetscape Navigatorなどのブラウザからアクセスした場合に，**画面１**のようなWebページをブラウザに返信する。ここでは一般ユーザ権限でサーバを実行させるために，TCPポート8080で簡易HTTPサーバプログラムを動作させている。そのためブラウザで指定するURLには，":8080"というTCPポート番号指定が必要である。またサンプルプログラムであるため，出力されるWebページはURLに関わらず固定である。

画面１　ブラウザから簡易 HTTP サーバ（ezhttpd.c）にアクセス



リスト１　簡易 HTTP サーバ ezhttpd.c

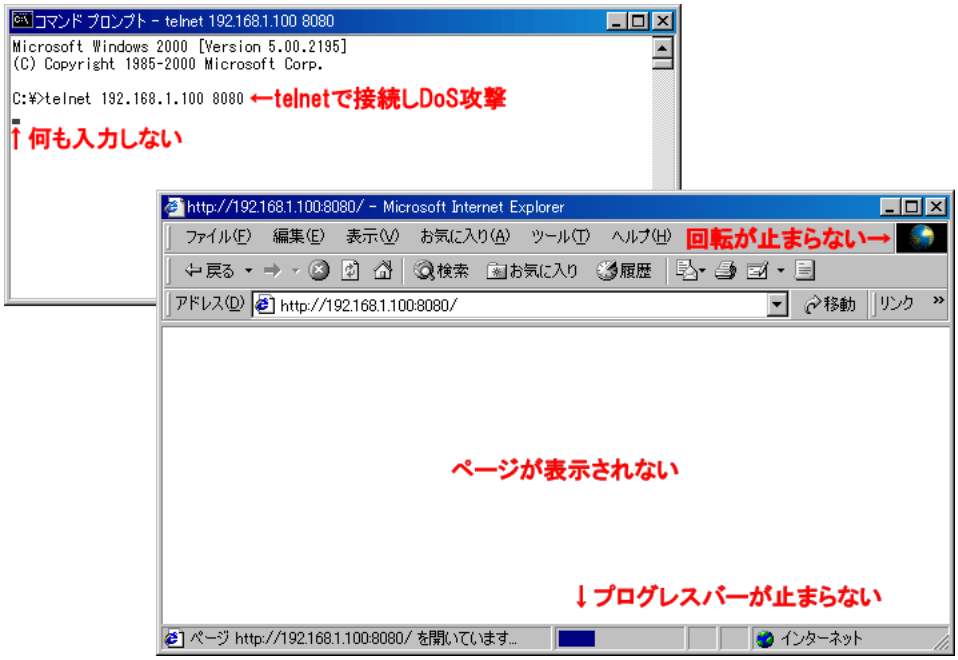
```
 1  #include "tcpserver.h"
 2  #include <stdio.h>
 3
 4  #define SERVER_PORT     8080            /* サーバ用 PORT */
 5  #define SERVER_IP       0x00000000UL    /* サーバ用待ち受け IP */
 6  #define BUFFER_SIZE     1024            /* バッファバイト数 */
 7
 8  /* prototypes */
 9  int main(void);
10  int do_httpd(TCPCLIENT_INFO* client);
11
12  /* メイン */
13  int main(void)
14  {
15      int ok = 1;             /* 1:処理続行 0:エラー発生 */
16      TCPSERVER_INFO server;
17      TCPCLIENT_INFO client;
18
19      /* サーバの初期化 */
20      if(ok) {
21          /* server 構造体初期化 */
22          tcpserver_init(&server, SERVER_IP, SERVER_PORT);
23
24          /* socket(), bind(), listen()をコール */
25          ok = tcpserver_open(&server);
26      }
27
28      if(ok) {
29          /* メインループ */
30          while(tcpserver_wait(&server, &client)) {   /* クライアント接続待ち */
31              /* クライアントから接続された */
32
33              /* HTTPD 処理 */
34              do_httpd(&client);
35
36              /* 後片付け：クライアントとの接続終了処理 */
37              tcpclient_close(&client);    /* FIN,ACK 送信 */
38          }
39      }
40
41      /* サーバの終了 */
42      tcpserver_close(&server);
43      return ok;
44  }
45
46  static const char *response_header =
47      "HTTP/1.0 200 OK\r\n"
48      "Server: ezhttpd(IPA Secure Programming Sample Program)\r\n"
49      "Content-Type: text/html\r\n"
50      "Content-Length: %d\r\n"
51      "Connection: close\r\n"
52      "\r\n";
53
54  static const char *response_body =
55      "<HTML>\r\n"
56      "<HEAD><TITLE>IPA Secure Programming - EZHTTPD</TITLE></HEAD>\r\n"
57      "<BODY>\r\n"
58      "<H1>IPA Secure Programming - EZHTTPD</H1>\r\n"
59      "This page is from the EZHTTPD sample web server.\r\n"
60      "</BODY>\r\n"
61      "</HTML>\r\n";
62
63  /* クライアントとの処理(HTTP)をつかさどる */
64  /* 返り値 1:正常終了 0:異常終了 */
65  int do_httpd(TCPCLIENT_INFO* client)
66  {
67      int     ok = 1;                 /* 正常終了時の返り値で初期化 */
68      char    req_buf[BUFFER_SIZE];
69      char    res_buf[BUFFER_SIZE];
70
71      /* リクエストを読み込む */
72      read(client->data_socket, req_buf, sizeof(req_buf));
73
74      /* レスポンスを送信 */
75      sprintf(res_buf, response_header, strlen(response_body));
76      write(client->data_socket, res_buf, strlen(res_buf));
77      write(client->data_socket, response_body, strlen(response_body));
78
79      return ok;
80  }
```

リスト１の簡易HTTPサーバは通常，**画面１**のように正しく動作する。しかし複数クライアントの同時接続を考慮していないため，DoS 攻撃の被害を受ける。**画面２**に DoS 攻撃の例を示す。

攻撃方法はいたって簡単で，簡易 HTTP サーバの TCP ポート 8080 へ Telnet 接続するだけである。**画面２**左上ではWindows上のコマンドプロンプトからTelnet接続を行っている。この状態でブラウザから簡易HTTPサーバへ接続を試みると，**画面２**右下のウィンドウのようにページがいつまで経っても表示されない。Telnet接続が1つ目のクライアント接続で，ブラウザからの接続が2つ目のクライアント接続となる。簡易HTTPサーバは同時に1つのクライアント接続しか処理できないため，1つ目のクライアント接続が終了しない限り，2つ目以降のクライアントへのサービスを提供できない。この状態が DoS 状態（サービス不能状態）である。

ここで左上のコマンドプロンプトで1文字タイプすると，1つ目のTelnet接続が終了し，右下のブラウザに**画面１**と同様のWebページが表示される。次節ではソースコードからこの一連の現象を説明する。

画面２　DoS 攻撃の例（左上ウィンドウ）と障害の発生（右下）



簡易 HTTP サーバのメインループ

リスト１の 30～38 行目のメインループ部分を見ていただきたい。HTTP サーバの動作を素直に反映した設計で，次の３つの手順の繰り返しループである。

手順１　クライアントからの接続を待つ（tcpserver_wait( )関数）
手順２　クライアントからの HTTP リクエストを受信し，HTTP レスポンスを返す（do_httpd( )関数）
手順３　クライアントとの接続を切断する（tcpclient_close( )関数）

一見問題が無いように見えるかもしれない。ここであるクライアント接続に対し，手順2の処理で非常に時間が掛かっている状況を想定しよう。**手順２**，**手順３**が終わるまでのしばらくの間，**手順１**まで処理が戻らないため，他のクライアントが接続できないという問題が生じる。もし**手順２**の処理が終わらない場合は，他のクライアントはいつまで経っても接続できなくなってしまう。前述の DoS 攻撃の例ではこの状況に陥っている。

手順２の do_httpd( )関数の実体は 65～80 行目にある。72 行目の read( )システムコールでクライアントからの HTTP リクエストを受信し，76～77 行目の write( )システムコールでクライアントへ HTTP レスポンスを返信

する。クライアントが接続してきたものの，HTTPリクエストをまったく送ってこない場合，サーバプロセスは72行目のread( )システムコールでブロックする。**画面２**のDoS攻撃の例ではTelnet接続によってこのようなクライアント接続を発生させている。

この状況でコマンドプロンプトから１文字タイプするとread( )システムコールは１バイト受信するため，ブロックしていたサーバプロセスは処理を再開する。出力するWebページのサイズは小さいため，続くwrite( )システムコールでブロックされることは無く，すぐにdo_httpd( )関数は終了する。**手順３**もプロセスがブロックする要因はないので，すぐに**手順１**に戻る。**手順１**に戻ることにより，他のクライアント接続を受け入れることができ，待たされていたクライアントへWebページが送信されることとなる。

この一連の処理の流れが**画面２**で示したDoS攻撃の現象の詳細である。このように複数クライアントの同時接続を考慮していないサーバプログラムは，同様なDoS脆弱性を持つ。

ネットワークサービスは必ずforkしよう

複数クライアントの同時接続に対応するため，fork( )システムコールを使うことができる。

プロセスがfork( )システムコールを呼び出すと，そのプロセスを複製したプロセスが生成される。この複製プロセスは呼び出しプロセスの子プロセスと呼ばれ，呼び出しプロセスはその親プロセスと呼ばれる。fork( )システムコールは呼び出しプロセスがオープンしているファイルディスクリプタも複製するため，親プロセスと子プロセスは，親プロセスがオープンしていたファイルやソケット，パイプなどを共有する。

fork( )システムコール呼び出しから戻る瞬間から，親プロセスと子プロセスが同時並行的に実行される。fork( )システムコールの返り値は親プロセスと子プロセスとで異なる。返り値を判断することにより，親プロセスにさせたい処理と子プロセスにさせたい処理を分けることができる。

親プロセスには子プロセスのプロセスIDが返され，正の値である。子プロセスには0が返される。またfork( )システムコールが失敗した場合は，親プロセスに-1が返され，子プロセスは生成されない。fork( )システムコールについては参考文献『詳細UNIXプログラミング［新装版］』や“man fork”（manはUnix/Linuxシステムに備わっているオンラインマニュアルを呼び出すコマンド）を参照していただきたい。

複数クライアントの同時接続に対応するため，**図１**に示すようなサーバ本体プロセスと子プロセスの役割分担を行う。サーバ本体プロセスはクライアントからの接続受け付け業務のみに専念する。クライアントから接続された場合に，サーバ本体プロセスはfork( )システムコールを呼び出し，子プロセスを生成する。子プロ

図１　サーバ本体プロセスと子プロセスの役割分担

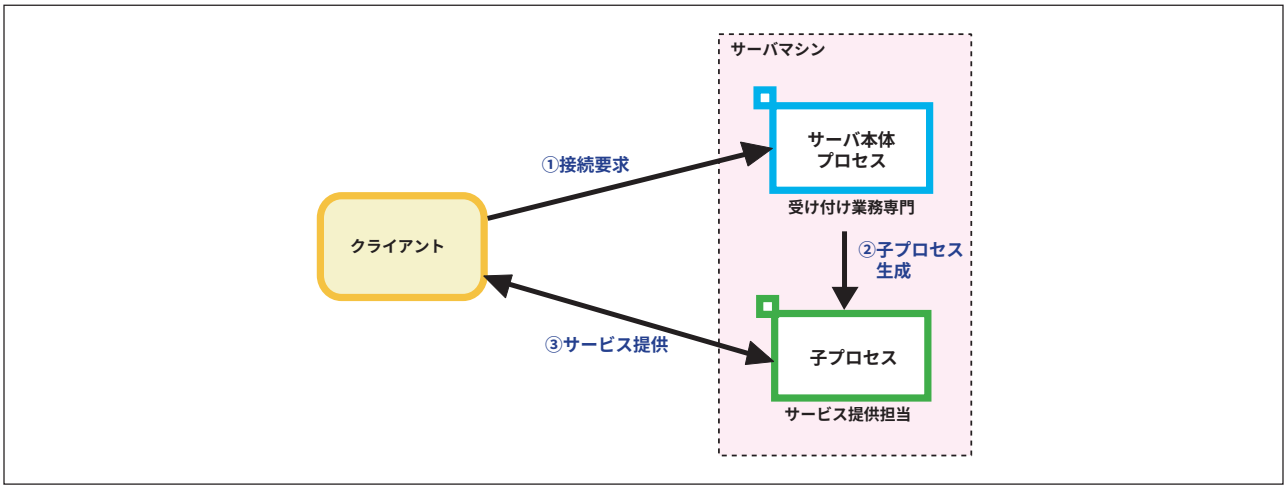
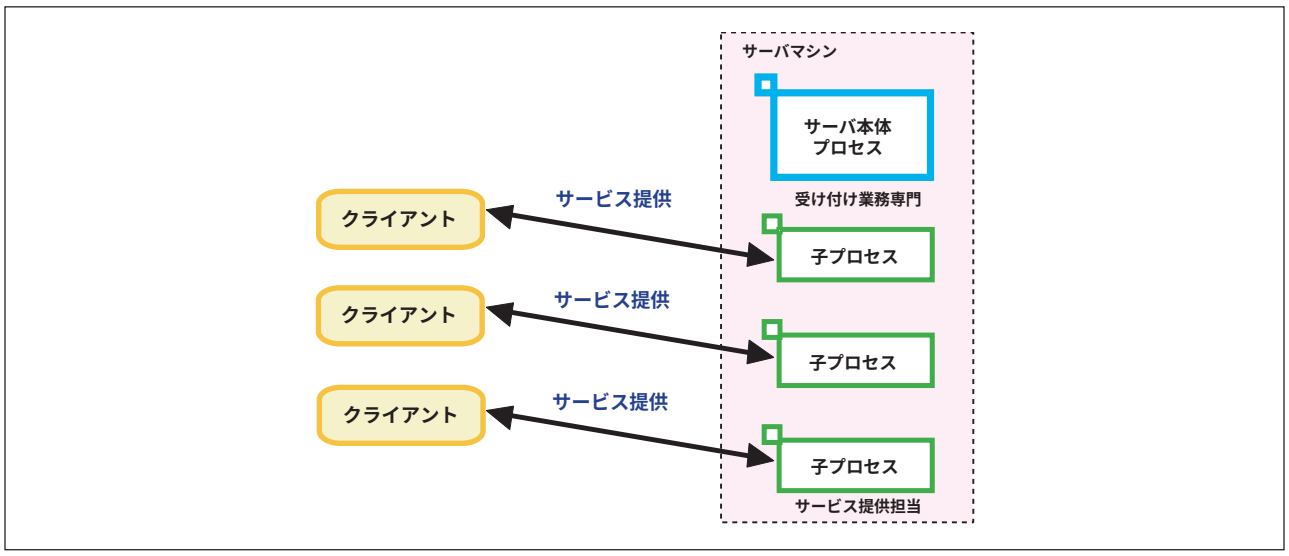


図２　クライアントと子プロセスの１対１関係



セスを生成した後は，もとの接続受け付け業務に戻る。生成された子プロセスは，サーバ本体プロセスが受け付けたクライアントに対しサービスを提供する。子プロセスはサービスが終了するまで存在し，サービスが終了すると自ら終了する。

図１に示したサーバ本体プロセスと子プロセスの役割分担を行うと，複数クライアント接続に対して**図２**のような関係ができる。子プロセスはクライアントの接続要求ごとに生成され，子プロセスとクライアントは１対１の関係を成す。１つの子プロセスは自分が担当する１つのクライアントへサービスを提供する。

この設計に従うと次のようなメリットがある。

- サーバ本体プロセスの処理は受け付け業務のみであるため，いつでもクライアントの接続要求に応答できる
- 子プロセスの処理は１つのクライアントを対象とすればよく，たとえクライアントの都合で子プロセスがブロックすることがあっても，他のクライアントへのサービスに影響を与えない

fork する簡易 HTTP サーバのメインループ

前節の設計に基づきfork( )システムコールを使用した，簡易HTTPサーバのサンプルプログラムを**リスト２**に示す。**リスト２**では**リスト１**に変更を加えた部分だけを示している。

32行目のtcpserver_wait( )関数はクライアントからの接続を受け付ける関数で，クライアントから接続されるまでサーバ本体プロセスをブロックする。クライアントから接続されると，サーバ本体プロセスは実行を再開し，36行目のfork( )システムコールを呼び出して子プロセスを生成する。fork( )システムコールの返り値をもとにif文で処理を分岐し，子プロセスに43～56行目を実行させ，サーバ本体プロセスに59～61行目を実行させている。

サーバ本体プロセスの処理が簡単なので先に説明する。tcpserver_wait( )関数でクライアント接続を受け付けると，そのクライアントとの全二重通信用ソケットが生成される。fork後，このソケットはサーバ本体プロセスと子プロセスで共有されているが，このソケットを通じてクライアントへサービスを提供するのは子プロセスであり，サーバ本体プロセスにとっては不要なソケットである。よって61行目でこのソケットをクローズしている。親プロセスで行うべき処理はこれだけであり，後は32行目のtcpserver_wait( )関数へ処理を戻し，新たなクライアント接続の待ち受け業務を再開する。

リスト２　fork する簡易 HTTP サーバ fork_ezhttpd.c

```
 2    #include <stdio.h>
 +01 #include <string.h>      /* memset() */
 +02 #include <unistd.h>      /* fork() */
 3

29          /* メインループ */
30          while(tcpserver_wait(&server, &client)) {   /* クライアント接続待ち */
31              /* クライアントから接続された */
32
  +01           /* 子プロセス生成 */
  +02           pid_t child_pid = fork();
  +03
  +04           if(child_pid==-1) {
  +05               /* fork()失敗→異常なのでサーバ終了 */
  +06               break;
  +07           }
  +08           else if(child_pid==0) {
  +09               /* 子プロセス */
  +10               int ret = 0;
  +11               /* 親プロセスのソケットを子プロセスに触らせたくないので
  +12                  クローズ */
  +13               tcpserver_close(&server);
  +14
33                  /* 子プロセスにやらせたい処理→HTTPD処理（1リクエスト分） */
34                  ret = do_httpd(&client);
35
36                  /* 後片付け：クライアントとの接続終了処理 */
37                  tcpclient_close(&client);    /* FIN,ACK送信 */
  +01
  +02               /* 子プロセスはここで自ら終了 */
  +03               exit(ret);
  +04           }
  +05           else {
  +06               /* 親プロセス */
  +07               /* 子プロセスに使わせるソケットは要らないのでクローズ */
  +08               tcpclient_close(&client);
  +09           }
38          }
```

次に子プロセスの処理について説明する。子プロセスはfork後，親プロセスのクライアント待ち受け用ソケットも共有しているので，47行目でこれをクローズしている。クライアントとの全二重通信用ソケットを通じて，50行目でHTTPサーバのサービス処理を行う。サービス処理後，子プロセスは53行目でクライアントとの全二重通信用ソケットをクローズし，56行目のexit( )システムコールで子プロセス自身を終了させる。もし**画面２**で例に挙げたようなDoS攻撃を受けた場合，50行目で呼び出すdo_httpd( )関数から戻ってこなくなる。しかし処理がブロックするのはこの子プロセスだけであり，サーバ本体プロセスや他の子プロセスは影響なく動作しつづける。**画面２**で示したようなDoS攻撃の被害はもはや発生しない。

ゾンビプロセスと wait( )システムコール

実は**リスト２**の設計は完全ではない。ゾンビプロセスが出来てしまう。**実行例１**に**リスト２**の簡易HTTPサーバを動作させているときのpsコマンドの出力例を示す。psコマンドは動作中のプロセスを一覧表示するUnixコマンドである。

3行目のfork_ezhttpdが簡易HTTPサーバの本体プロセスである。4～7行目にfork_ezhttpd <defunct>と表示されているプロセスは，リスト2における終了後の子プロセスであり，これがゾンビプロセスである。子プロセスは終了すると消滅せず，ゾンビプロセスとして残りつづけているのである。ブラウザでリロードするたびに，このゾンビプロセスは増えつづけていく。一般に，システムやユーザ単位のプロセス数には上限があるため，いずれfork( )システムコールで失敗するようになる。

Unixプロセスは終了すると，一旦ゾンビプロセスとなる。その後，そのプロセスの親プロセスがwait( )システムコールを呼び出すまで存在しつづける。親プロセスがwait( )システムコールを呼び出すと，ゾンビプロセ

実行例１　ゾンビプロセス

```
1  $ ps -a
2    PID TTY          TIME CMD
3    674 pts/0    00:00:00 fork_ezhttpd
4    683 pts/0    00:00:00 fork_ezhttpd <defunct>
5    686 pts/0    00:00:00 fork_ezhttpd <defunct>
6    688 pts/0    00:00:00 fork_ezhttpd <defunct>
7    689 pts/0    00:00:00 fork_ezhttpd <defunct>
8    696 pts/0    00:00:00 ps
9  $
```

スの終了時の終了ステータスが親プロセスに返され，同時にゾンビプロセスも消滅する。このように，子プロセスの終了ステータスを親プロセスに伝える仕組みとして，ゾンビプロセスが存在する。

あるプロセスがwait( )システムコールを呼び出すと，その子プロセスが既に終了している場合は直ちに終了ステータスを返す。しかしどの子プロセスも終了していない場合，wait( )システムコールを呼び出したプロセスは，いずれかの子プロセスが終了するまでブロックしてしまう。よって**リスト２**のサーバ本体プロセスのクライアント接続受け付けループでwait( )システムコールを呼び出すわけには行かない。子プロセスが終了するまで wait( )システムコールはブロックしてしまい，受け付け業務ができなくなるからである。

SIGCHLD シグナル

子プロセスの終了タイミングは親プロセスによって予測不可能であることが多い。よって親プロセスがwait( )システムコールを呼び出すべきタイミングが難しい。これを補うために，子プロセスが終了したタイミングで親プロセスに対して送られるシグナルがある。SIGCHLD シグナルである。

シグナルはUnixのソフトウェア割り込み機構で非同期処理の実装に利用される。あるプロセスがシグナルを受信すると，そのプロセスが現在実行している処理を一旦中断し，そのシグナルに対応したシグナルハンドラを実行する。シグナルハンドラの処理が終了すると，もとの処理を再開する。シグナルハンドラはプロセスがシグナルを受信したときに実行する関数である。シグナルについては参考文献『詳細UNIXプログラミング［新装版］』や“ man signal ”（Unix/Linux システムのオンラインマニュアル）を参照していただきたい。

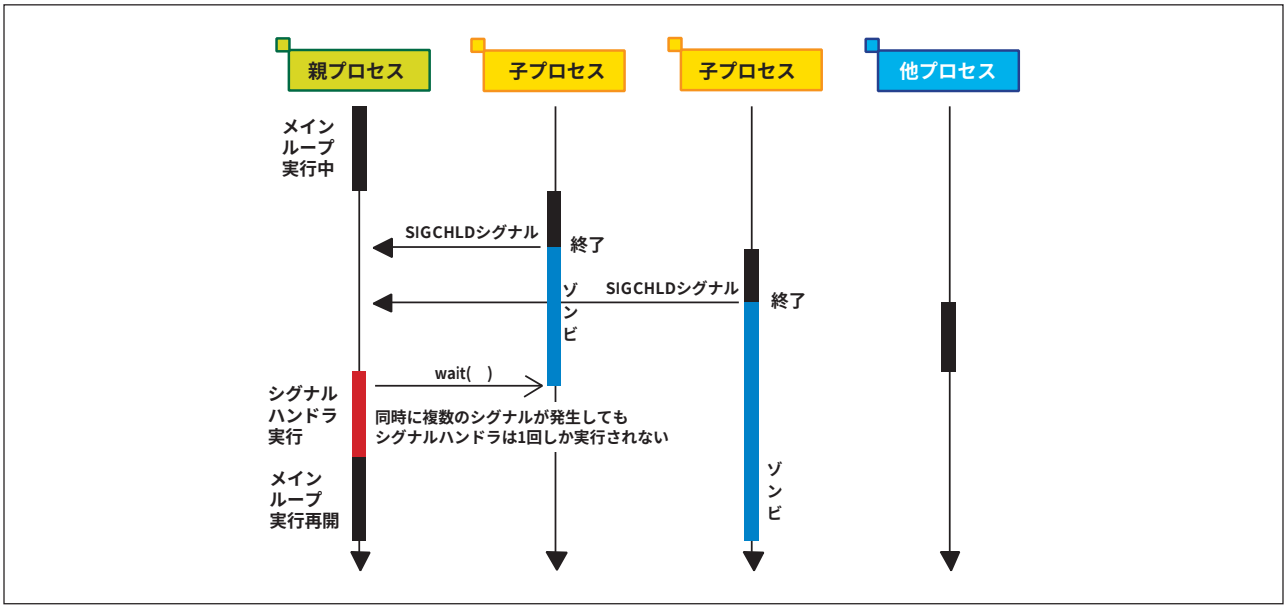
SIGCHLDシグナルに割り当てるシグナルハンドラ関数にて，wait( )システムコールを呼び出すことで，子プロセスが終了したタイミングで親プロセスがwait( )システムコールを呼び出すことができる。これにより子プロセス終了後のゾンビプロセスは適切に削除される。

シグナルの取りこぼしと waitpid( )システムコール

SIGCHLDシグナルのシグナルハンドラでwait( )システムコールを呼び出すだけでは，実は不十分である。Unixプロセスのタイムシェアリングにより発生しうる状況として**図３**のような場合がある。

図３は1 CPUシステムで同時に1プロセスのみがCPU使用権を持つタイムシェアリングシステムの例である。この例では２つの子プロセスが終了し，２つの SIGCHLD シグナルが発生しているが，１回しかシグナルハンドラが実行されていない。これはUnix シグナルの仕様である。シグナルハンドラで wait( )システムコールを1回呼び出しただけでは，1つ目のゾンビプロセスしか処理できない。2つ目のゾンビプロセスはそのままゾンビプロセスとして残ってしまう。

図３　シグナルの取りこぼし



したがってシグナルハンドラでは，ゾンビプロセスの個数に応じて，wait( )システムコールを複数回呼び出さなくてはならない。ところが親プロセスにはゾンビプロセスの個数を知る手段がないため，wait( )システムコールを何回呼び出せばよいのか分からない。またwait( )システムコールを余分に呼び出してしまうと，親プロセスはブロックしてしまうため，シグナルハンドラ内で親プロセスの処理が停止してしまうことになる。

こうした問題に対応できるよう waitpid( )システムコールが導入された。waitpid( )システムコールは wait( )システムコールの言わば高機能版である。waitpid( )システムコールについては，参考文献『詳細UNIXプログラミング［新装版］』や“ man waitpid ”（Unix/Linux システムのオンラインマニュアル）を参照していただきたい。waitpid( )システムコールの第3引数にWNOHANGオプションを指定することにより，プロセスがブロックしないようになる。waitpid( )システムコールはゾンビプロセスを処理した場合，そのゾンビプロセスのプロセス ID を返すが，ゾンビプロセスがいない場合は 0 を返す。この WNOHANG オプションの特性を利用することにより，SIGCHLD シグナルのシグナルハンドラで，waitpid( )システムコールを返り値が0になるまで繰り返すことにより，すべてのゾンビプロセスに対しwaitpid( )システムコールを呼び出すことができる。これが SIGCHLD シグナルの完全なシグナルハンドラの設計方法である。

ゾンビプロセス対策を施した簡易 HTTP サーバ

ゾンビプロセス対策を施した簡易HTTPサーバのサンプルプログラムを**リスト３**に示す。**リスト３**は**リスト２**に変更を加えた部分のみを示してある。

サーバ初期化部分の29行目にsetup_SIGCHLD( )関数の呼び出しを追加している。この関数の実体は115～126行目にある。122行目でシグナルハンドラ関数として128行目から始るcatch_SIGCHLD( )関数を指定している。125 行目の sigaction( )システムコール呼び出しにより，SIGCHLD シグナルが発生したときに catch_SIGCHLD( )関数が呼ばれるようになる。これ以降，子プロセスが終了するタイミングで，catch_SIGCHLD( )関数が親プロセスにより実行されるようになる。

catch_SIGCHLD( )関数における重要な処理は，135 行の waitpid( )システムコール呼び出しである。133 ～ 138 行の do while ループにて，0 を返してくるまで waitpid( )システムコールを WNOHANG オプションを指定して繰り返し呼び出している。これによりSIGCHLDシグナルの取りこぼしが発生した場合にも，すべてのゾンビプロセスを適切に処理している。

リスト３　ゾンビプロセス対策を施した簡易 HTTP サーバ signal_ezhttpd.c

```
 19      /* サーバの初期化 */
 20      if(ok) {
   +01       /* 子プロセス終了時のシグナルハンドラを設定 */
   +02       setup_SIGCHLD();
   +03
 21          /* server 構造体初期化 */
 22          tcpserver_init(&server, SERVER_IP, SERVER_PORT);
 23
 24          /* socket(), bind(), listen()をコール */
 25          ok = tcpserver_open(&server);
 26      }

 81
 82  /* 子プロセス終了時の処理 - setup_SIGCHLD(), catch_SIGCHLD()
 83      子プロセスが終了した場合に，SIGCHLD シグナルを捕捉し，waitpid()を
 84      呼んであげないと，子プロセスはゾンビプロセスとして生きつづける。 */
 85  void setup_SIGCHLD()
 86  {
 87      struct sigaction act;
 88      memset(&act, 0, sizeof(act));   /* sigaction 構造体をとりあえずクリア */
 89      act.sa_handler = catch_SIGCHLD; /* SIGCHLD 発生時に catch_SIGCHLD()を実行 */
 90      sigemptyset(&act.sa_mask);  /* catch_SIGCHLD()中の追加シグナルマスクなし */
 91      act.sa_flags = SA_NOCLDSTOP | SA_RESTART;
 92      sigaction(SIGCHLD, &act, NULL);
 93  }
 94
 95  void catch_SIGCHLD(int signo)
 96  {
 97      pid_t child_pid = 0;    /* とりあえず初期化 */
 98
 99      /* すべての終了している子プロセスに対して waitpid()を呼ぶ */
100      do {
101          int child_ret;
102          child_pid = waitpid(-1, &child_ret, WNOHANG);
103          /* すべての終了している子プロセスへ waitpid()を呼ぶと
104             WNOHANG オプションにより waitpid()は 0 を返す */
105      } while(child_pid>0);
106  }
```

まとめ

複数クライアントの同時接続を考慮していないサーバプログラムはDoS脆弱性を有する可能性がある。fork( )システムコールを使って複数クライアントの同時接続を実装しよう。子プロセスが終了した際に生じるゾンビプロセスにはSIGCHLDシグナルをキャッチしてwaitpid( )システムコールで対処するのを忘れてはならない。イントラネット内で使用するだけのちょっとしたネットワークサービスであっても，本稿で紹介したテクニックを適用して，より安全なネットワークサービスを実現しよう。

参考文献

『詳細UNIX プログラミング［新装版］』，W・リチャード・スティーヴンス，大木 敦雄訳，2000 年，株式会社ピアソン・エデュケーション

“ man fork ”, “ man signal ”, “ man waitpid ”
（Unix / Linux システムのオンラインマニュアル）

リスト４　TCP サーバ用ライブラリ tcpserver.h

```
 1  /*
 2   * tcpserver.h
 3   *
 4   */
 5
 6  /*
 7   * 使い方
 8   *
 9   *  TCPSERVER_INFO server;
10   *  tcpserver_init(&server, 0, 1234);           ポート 1234 でサービス提供
11   *  tcpserver_open(&server);                    ポート 1234 を bind(), listen()
12   *  for(;;) {
13   *      TCPCLIENT_INFO client;
14   *      tcpserver_wait(&server, &client);       クライアント接続を待つ
15   *      spawn_child_process(&server, &client);  子プロセスを生成し
16   *  }                                           クライアントと通信させる
17   *
18   *  spawn_child_process()関数で, fork()などして,
19   *  client->data_socket に対して read(), write()することで,
20   *  クライアントと通信する
21   *
22   *  サンプルはこちら
23   *
24   *  void spawn_child_process(TCPSERVER_INFO *server, TCPCLIENT_INFO *client) {
25   *      int child_pid = fork();
26   *      switch(child_pic) {
27   *      case -1:    ← fork()でエラー発生
28   *          return;
29   *      case 0:     ← 子プロセスの処理
30   *          close_parent_fds();
31   *          exit(do_service_for_client(client));
32   *          break;
33   *      default:    ← 親プロセスは何もせず
34   *          tcpclient_close(client);
35   *          return; ← そのままリターン
36   *       }
37   *  }
38   */
39
40  #ifndef _TCPSERVER_H_
41  #define _TCPSERVER_H_
42
43  #include <sys/socket.h>
44  #include <netinet/in.h>
45  #include <stdio.h>
46
47  /* クライアントからの接続を待つサーバの情報を表現する構造体 */
48  typedef struct {
49      int                 wait_socket;    /* サーバ待ち受け用ソケット */
50      struct sockaddr_in  server_addr;    /* サーバ待ち受け用アドレス */
51  } TCPSERVER_INFO;
52
53  /* クライアントとの接続に関する情報を保存する構造体 */
54  typedef struct {
55      int                 data_socket;    /* クライアントとの通信用ソケット */
56      struct sockaddr_in  client_addr;    /* クライアントのアドレス */
57  } TCPCLIENT_INFO;
58
59  /* TCPSERVER_INFO 構造体の初期化 */
60  extern void tcpserver_init(
61      TCPSERVER_INFO* server_info,    /* ←この構造体を初期化する */
62      unsigned long   ip,             /* サーバ待受 IP アドレス（32 ビット値） */
63      unsigned short  port);          /* サーバ待受ポート番号 */
64
65  /* サーバ側ソケットの作成・設置 */
66  extern int tcpserver_open(          /* 帰り値 1:成功 0:エラー */
67      TCPSERVER_INFO* server_info);
68
69  /* サーバ側ソケットの開放 */
70  extern void tcpserver_close(
71      TCPSERVER_INFO* server_info);
72
73  /* クライアントからの接続を待機・確立 */
74  /* 帰り値 1:クライアントとの接続を確立した 0:エラー */
75  extern int tcpserver_wait(
76      TCPSERVER_INFO* server_info,    /* tcpserver_init()で初期化したもの */
77      TCPCLIENT_INFO* client_info);   /* 接続してきたクライアントの情報が入る */
78
79  /* クライアントとの接続を終了 */
```

リスト４（つづき）

```
80  extern void tcpclient_close(
81      TCPCLIENT_INFO* client_info);   /* tcpserver_wait()で初期化されたもの */
82
83  #endif
```

リスト５　TCP サーバ用ライブラリ tcpserver.c

```
 1  /*
 2   * tcpserver.c
 3   *
 4   */
 5
 6  #include "tcpserver.h"
 7  #include <stdio.h>
 8  #include <sys/types.h>
 9  #include <sys/socket.h>
10  #include <netinet/in.h>
11  #include <strings.h>
12  #include <errno.h>
13
14  /* TCPSERVER_INFO構造体の初期化 */
15  void tcpserver_init(
16      TCPSERVER_INFO* server_info,
17      unsigned long   ip,
18      unsigned short  port)
19  {
20      /* まずはゼロクリア */
21      memset(server_info, 0, sizeof(*server_info));
22
23      /* socket()がエラーのとき-1を返すことを考慮し-1と初期化 */
24      /* エラー後にtcpserver_close()でソケットクローズしないための工夫 */
25      server_info->wait_socket = -1;
26
27      /* IPアドレスとポート番号をセット */
28      server_info->server_addr.sin_family     = AF_INET;      /* TCP/IP */
29      server_info->server_addr.sin_addr.s_addr= htonl(ip);    /* IPアドレス */
30      server_info->server_addr.sin_port       = htons(port);  /* ポート番号 */
31  }
32
33  /* サーバ待ち受け用ソケットの作成・設置 */
34  /* 帰り値 1:成功 0:エラー */
35  int tcpserver_open(TCPSERVER_INFO* server_info)
36  {
37      /* TCPソケットを作成 */
38      server_info->wait_socket = socket(PF_INET, SOCK_STREAM, 0);
39      if(server_info->wait_socket==-1) {
40          /* ソケット作成の失敗 */
41          perror("tcpserver_open.socket");
42          return 0;
43      }
44
45      /* TCPソケットを設置 */
46      /* これによりサーバ側のポート（クライアント待ち受け用）を占有する */
47      /* このときTCPソケットの状態はクローズ（CLOSE）となる */
48      if(bind(server_info->wait_socket,
49              (struct sockaddr*)&server_info->server_addr,
50              sizeof(server_info->server_addr))==-1)
51      {
52          /* ソケット設置の失敗 - 主に以下の2つが原因 */
53          /* (1)ポートが既に使用されている */
54          /* (2)ポート番号が1023以下であるにも関わらず一般ユーザ権限で実行 */
55          perror("tcpserver_open.bind");
56          return 0;
57      }
58
59      /* TCPソケットをクライアント接続待ち状態（LISTEN）にする */
60      if(listen(server_info->wait_socket, 5)==-1) {
61          /* 普通はこんなところでは失敗しない→異常？ */
62          perror("tcpserver_open.listen");
63          return 0;
64      }
65
66      /* 正常終了 */
67      return 1;
```

リスト５（つづき）

```
 68  }
 69
 70  /* サーバ待ち受け用ソケットの開放 */
 71  void tcpserver_close(TCPSERVER_INFO* server_info)
 72  {
 73      /* ソケットがオープンされているならクローズする */
 74      if(server_info->wait_socket != -1) {
 75          close(server_info->wait_socket);
 76          server_info->wait_socket = -1;
 77      }
 78  }
 79
 80  /* クライアントからの接続を待機・確立 */
 81  /* 帰り値 1:クライアントとの接続を確立した 0:エラー */
 82  int tcpserver_wait(
 83      TCPSERVER_INFO* server_info,    /* tcpserver_init()で初期化したもの */
 84      TCPCLIENT_INFO* client_info)    /* 接続してきたクライアントの情報が入る */
 85  {
 86      /* client_info 構造体を初期化 */
 87      memset(client_info, 0, sizeof(*client_info));   /* ゼロクリア */
 88      client_info->data_socket = -1;      /* accept()のエラー値で初期化 */
 89
 90      /* クライアント接続を待機し，接続されたらデータ送受用ソケットを取得 */
 91      /* for(;;)ループはシグナル受信時に再accept()するため必要 */
 92      for(;;) {
 93          socklen_t client_addr_len = sizeof(client_info->client_addr);
 94          client_info->data_socket =
 95              accept(server_info->wait_socket,
 96                     (struct sockaddr*)&client_info->client_addr,
 97                     &client_addr_len);
 98
 99          /* accept()の返り値チェック */
100          if(client_info->data_socket!=-1) break; /* 正常終了 */
101          if(errno==EINTR) continue;              /* シグナル受信→再accept() */
102          perror("tcpserver_open.accept");        /* やっぱりエラー */
103          return 0;   /* エラー終了 */
104      }
105
106      /* 正常終了 */
107      return 1;
108  }
109
110  /* クライアントとの接続を切断 */
111  void tcpclient_close(
112      TCPCLIENT_INFO* client_info)    /* 接続してきたクライアントの情報が入る */
113  {
114      /* ソケットがオープンされているならクローズする */
115      if(client_info->data_socket != -1) {
116          close(client_info->data_socket);                /* ソケットクローズ */
117          client_info->data_socket = -1;
118      }
119  }
```