サーバー構築/運用術

第4回 Perl(後編)

先月号に引き続き，perlを解説します。昨今のディストリビューションは複雑すぎて本質が見えにくくなっています。perlを使って一部を書き直してみることにより，サーバー・システムに対する理解がより深まることでしょう。一例として，syslogの監視とppp接続を行うperlスクリプトを紹介します。

(ケイ・ラボラトリー　仙石 浩明)

先月号に引き続き，昔話をします*1。私が最初に購入したコンピュータは，シャープのMZ-80K(**写真1**)シリーズの最後の機種K2E*2でした。RAMは32Kバイト，CPUは当時流行だったZilog Z-80 2MHz*3で，BASICインタプリタをROMではなく，カセット・テープから読み込む方式であるため，BASIC以外の言語への対応が容易という，当時としては画期的*4なパソコンでした。BASICライクなアセンブラBASE-80*5を使って，ゲームやモニター*6を作っていたことを思い出します。その後，CP/Mを移植するためにBIOSを記述する際にも，BASE-80を使いました。

今日の基準からすれば，恐ろしく貧弱な開発環境で，実際ろくなプログラムは作れなかったのですが，コンピュータの基礎を学ぶには最適の環境でした。ROM（SP-1002）はわずかに4Kバイトだったので，すべて逆アセンブルして解析することができましたし，ハードウエアはCPUなどのLSIを除けばTTLなどのICで大部分が構成されていて，基板の配線を追って回路図を再構成し，仕組みを理解することもできました。そして，CPUクロックの倍速スイッチを付けたり，キャラクタしか表示できなかったディスプレイにグラフィックスを表示できるように改造したり，Z-80を使ったコンピュータを手作り*7していたものです。

当時高校生で，論理回路のイロハすら知らなかった*8私にも設計できたのは，ひとえに当時のパソコンが単純だったからです。その後，パソコンは急速に複雑化・高度化し，初心者には手も足も出ない*9ものになってしまったのは残念なことです。

写真1　シャープが1978年12月に発売したMZ-80K
後継モデルのMZ-80K2Eは，「MZクリーンコンピュータ10万台突破」を記念して発売された。MZ-80KとMZ-80K2Eの外観はほとんど同じだが，ディスプレイのきょう体の色(MZ-80Kは赤，MZ-80K2Eは深緑)やキーボードの色が異なる。

何を学ぶにしても同じだと思うのですが，学ぶ対象は単純であればあるほど望ましいものでしょう。現在のPCの

＊1　年をとった証拠ですね。会社の同僚に，先月号に載せたPET2001の写真を見せたら，「コモドール？何それ？1978年？当時4歳でしたよ」などと言われてしまいました。
＊2　定価は14万8000円。キーボードが格子状のレイアウトであり，カセット・デッキが標準装備されている点などが，私が初めて触ったパソコンであるPET 2001と良く似ていたことから引かれたのでしょう。本当はMZ-80Bが欲しかったのですが，定価が27万8000円と高価で，とても手が届きませんでした。
＊3　20MHzの誤植ではありません。
＊4　電源を入れるたびに，1200bpsのカセット・テープからBASICインタプリタを読み込まなければならず，BASIC全盛の当時は少々無理があり，あまり売れなかったようです。フロッピ・ディスク(当時は5インチ)が普及して，インタプリタ等はディスクから読み込むのが当たり前になるのは数年後のことです。
＊5　キャリー・ラボ製。BASICのような代入文や複文を使うことができたアセンブラです。雑誌に掲載されていたBASE-80の16進ダンプを，自作の16進キーボードを使って(文字通り1バイトずつ)入力しました。あらゆる種類のフリーソフトウエアがインターネットから容易に入手できる今日からは想像もできない時代です。
＊6　メモリーの16進ダンプを行ったり，カセット・テープから任意のメモリー番地へプログラムを読み込んだり，逆にカセット・テープへ出力したり，任意の番地へジャンプさせたりするためのプログラムです。

ハードウエアは，論理回路を学ぶには複雑すぎますし，現在のサーバー・システムは，サーバー運用管理を学ぶには複雑すぎます。

ディストリビューション

Linuxも，登場当時は単純でした。Linuxが普及し始めた初期*10のころのディストリビューションでは，ブート・スクリプト*11の他は各プログラムごとの設定ファイルがあるだけで，それぞれのプログラムを1つずつ理解していけば，システム全体が理解できたものです*12。

ところが，Linuxの普及にともない，初心者にも管理できるようにと管理ツールのGUI化が進みました。確かに見た目はマウスで操作できるようになり，とっつきやすくなったのですが，見えない部分は逆に複雑になってしまいました。つまり初心者が遭遇するあらゆる事態に対処できるようにするため，そして簡単な操作で適切な設定*13ができるようにするため，システム全体を統一的に管理する複雑な仕掛けが構築されていったのです。

各プログラムは互いに密接に連携するようになってしまい，ある設定変更の影響がどこまで及ぶかは，システム全体を完全に理解しない限り，把握できなくなってしまいました。サーバー運用管理を学ぼうとする管理者にとって，非常に不適切な環境と言わざるを得ません。

とは言っても，現実に管理すべきサーバーがある場合，インストール済みのシステムをすべて捨てて単純なディストリビューションからやり直すわけにはいかないでしょう。理解できていなくてもGUI管理ツールの助けで一通り設定はできるし，ある程度の管理はできているわけですから。

そこでお勧めしたいのが，ディストリビューションへの依存を徐々に減らしていく方法です。まず現時点で使用していないパッケージを削除します。Red Hat系のディストリビューションであれば，rpmコマンドを使ってパッケージを一覧し(**図1**)，要らなそうなパッケージの内容を調べ(**図2**)，削除します(**図3**)。もしそのパッケージを必要としているパッケージが他にあるのなら(**図4**)，そちらのパッケージについても必要か否か判断することになります。

これからLinuxシステムをインストールしようとしているのであれば，必要最小限のパッケージのみをインストールし，すぐ使う予定のないパッケージはインストールしないよう心掛けてください。

そして，何か必要なプログラムがあるときは，パッケージを入手してrpmコマンドを使ってインストールするのではなく，そのプログラムの開発元が公開しているソース・プログラムを入手し，必ず付属の説明書を熟読した上で適切な設定を行ってmakeし，インストールするようにしてください。/usr/localディレクトリ以下にインストールするようにすれば，既存のパッケージとの競合を最小限に抑えることができるでしょう。

あるパッケージをバージョンアップしようとする場合も，可能であればまずそのパッケージを削除した後に，ソース・プログラムからmakeして/usr/localディレクトリ以下にインストールするようにしてください。多くの場合，パッケージを削除しようとしてもそのパッケージを必要とする関連パッケージがあって，削除することができないのですが，関連パッケージの内容がすべて把握できて，それらを削除したときの影響が予想できるのであれば，思いきって全部削除し，必要なものだけソース・プログラムからmakeしてインストールしましょう。

このようにして次第にディストリビューションへの依存度を減らしていけば，

＊7　もちろん，今日のPC組み立てとは意味が異なり，ICの足を1本づつワイヤーで半田付けしながら組み立てました。
＊8　大学に入って初めて論理回路設計の基礎を学んだとき，設計にかかる時間が，我流の設計手法に比べて1/10以下で済むことを知って，まさに目からうろこ状態でした。
＊9　今では古い規格となってしまった33MHzのPCIバスでさえ，素人が手作りするには動作周波数が高すぎます。
＊10　1993年ごろのことです。Softlanding SoftwareやYggdrasil Computingがディストリビューションを公開していました。
＊11　Linuxシステムがブートするときに実行されるスクリプト。/etc/rc.d/rc.Sなど。詳しくは2001年3月号，pp.110-120の「実戦で学ぶ，一歩進んだサーバー構築術」を参照してください。
＊12　教育用の単純なディストリビューションがあるといいのかも知れません。どなたか作ってみませんか？
＊13　セキュリティが重視されるようになって以来，あらゆる環境下である程度の安全性を確保するために，ますます複雑になってしまいました。

```
# rpm -qa↵
setup-2.0.5-2k1
filesystem-1.3.5-1
basesystem-6.0-4
ldconfig-1.9.5-16
glibc-devel-2.1.2-17k5
        ...
```

図1 パッケージ一覧を表示
↵このマークで改行

```
# rpm -qi mouseconfig↵
Name        : mouseconfig                  Relocations: (not relocateable)
        ...
Summary     : The Red Hat Linux mouse configuration tool.
Description :
Mouseconfig is a text-based mouse configuration tool.  Mouseconfig
sets up the files and links needed for configuring and using a mouse
on a Red Hat Linux system.  The mouseconfig tool can be used to set
the correct mouse type for programs like gpm, and can be used with
Xconfigurator to set up the mouse for the X Window System.
```

図2 パッケージの内容を調べる

```
# rpm -e mouseconfig↵
```

図3 パッケージを削除する

```
# rpm -e truetype-fonts-ja↵
error: removing these packages would break dependencies:
        truetype-fonts-ja is needed by VFlib-2.24.2-5k4
```

図4 このパッケージを必要とする他のパッケージがある場合

自分で内容を理解した上でインストールしたプログラムが大半になるでしょうから，システム全体の動きも自然と見えてくることでしょう。ある設定ファイルを変更したら，意図しなかったプログラムにまで影響が及んだ，などということは防ぐことができるようになるはずです。

perlスクリプト

システムをある程度理解できるようになってくると，システムに対する要求もより具体的なものになることでしょう。例えば，「セキュリティ対策を行いたい」という漠然とした願望は，「syslog監視を自動化したい」などのはっきりした目的になります。

目的に応じて，必要なプログラムを探し出してインストールしても良いのですが，常に希望通りのプログラムが見つかるとは限りませんから，サーバー管理者を目指す読者の皆さんには，ぜひご自身でプログラムを作ってみることをお勧めします。実際にプログラムを作ることによりシステムに対する理解がより深まる，というメリットもあります。

一般に，プログラムは汎用であればあるほど複雑になり，逆に特定の環境および特定の用途向けのものは単純になります。自分だけのごく限られた範囲の目的にのみ使用するプログラムであれば，簡単なプログラムで済んでしまう場合が多いものです。そして，そういったプログラムを記述するのに最適なのが，万能スクリプト言語であるperlです。

ここでは，syslogの監視と，ppp接続を例にperlスクリプトの書き方を紹介します。実を言えば，syslog監視を行うプログラムとしてはswatch[*14]が有名ですし，ppp接続のためのツールとしてRed Hat系ディストリビューションにはnetcfgが標準で用意されていますが，出来合いのツールを使うよりは，自分でツールを作る方が勉強になります。トラブルが起きたときも自作のスクリプトならばすぐに修正可能ですし，目的に合わせて自由に機能を追加していくことができるようになります。

syslogの監視

syslogの出力を監視するツールの場合，サーバーの運用形態によって注目すべきログは異なりますから，個々のサーバーの事情に合わせて作り込んだperl

＊14 「The Simple WATCHer」(http://www.engr.ucsb.edu/~eta/swatch/)を参照してください。

```
#!/usr/bin/perl
while(<>) {
    next if /^\w+\s+\d+\s+\d+:\d+:\d+\s+\w+\s+-- MARK --$/;
    $log = $_;
    if (! /^\w+\s+\d+\s+\d+:\d+:\d+\s+(\w+)\s+([^:\[ ]+)(\[\d+\])?:?\s+(.*)\n/) {
        print "UNKNOWN $log";
        next;
    }
    $host = $1;
    $name = $2;
    $mesg = $4;
    if ($name =~ /^syslogd/) {
        next if $mesg eq "restart.";
    }
    elsif ($name eq "sshd") {
        my($user);
        if ($mesg =~ /^Accepted rsa for (\w+) from (\d+\.\d+\.\d+\.\d+) port \d+$/) {
            if ($host eq "asao") {
                next if $1 eq "sengoku" && $2 eq "192.168.1.2";
            }
        }
        if ($mesg =~ /^log: Connection from (\d+\.\d+\.\d+\.\d+) port \d+$/ ||
            $mesg =~ /^log: Closing connection to (\d+\.\d+\.\d+\.\d+)$/) {
            next if $1 eq "127.0.0.1";
            next if $1 eq "192.168.1.2";
        }
        next if $mesg eq "Generating new 768 bit RSA key.";
        next if $mesg eq "RSA key generation complete.";
    }
    print $log if $mesg ne $prev_mesg;
    $prev_mesg = $mesg;
}
```

図5　syslogの出力から重要でない行を取り除くスクリプト

```
May 13 02:54:44 asao -- MARK --
```

図6　ログがないときにsyslogが出力するマーカー

スクリプトの方が，既存のツールよりも適していると言えるでしょう。また，ツールを作っていく過程でsyslogの出力をじっくり観察することになりますから，システムを理解する上でも有益です。

syslogの監視方法としては，次の2通りが考えられます。

(1) syslogの出力の中に，あらかじめ設定しておいた条件を満たす行が見つかれば管理者に知らせる。

(2) syslogの出力から，あらかじめ設定しておいた条件を満たす行を削除し，残った未知の行を管理者に知らせる。

一般的に侵入方法を予測することは困難です。そもそも予測できる侵入方法ならわざわざ穴を放置せず，侵入できないように対策すべきですから，(1)はあまり適切な方法とは言えないでしょう。そこでここでは(2)の方法を考えます。つまり，大量のsyslogの出力の中から，あまり重要でない，すなわち次の条件を満たす行を取り除き，残ったものを管理者に知らせることになります。

- **侵入などの危険が少ない**
- **出現頻度が高い**

スクリプトの例を**図5**に示します。このスクリプトは標準入力からログを入力し，上記条件を満たす行を取り除き，残りを標準出力へ出力します。実際の運用では標準出力を管理者に知らせる仕掛けを追加する必要があるでしょう。

図5の2行目で，標準入力から1行ずつ読み込み，4行目で変数$logへ代入します。3行目は，ログが無いときにsyslogが自動的に出力する**図6**のような行を取り除きます。ここで「\w」「\s」「\d」は，それぞれアルファベット1文字，空白1文字，数字1文字を表わす正規表現です。perlで使用可能な，特定の種類の1文字にマ

ッチする正規表現を**表1**にまとめておきます。「\w」などに続く「+」は直前の正規表現の1回以上の繰り返しを意味する修飾子です。修飾子の一覧を**表2**に示します。また，最初の「\w」の前にある「^」は，文字列にマッチさせるとき，この位置が文字列の先頭であることを意味する正規表現です。同様に文字列中の位置を指定する正規表現を，**表3**に示します。つまり，3行目の正規表現

```
^\w+\s+\d+\s+\d+:\d+:\d+\s+
\w+\s+-- MARK --$
```

は，最初の「\w+」が図6の「May」にマッチし，以下「\s+」が「 」，「\d+\s+」が「13 」，「\d+:\d+:\d+\s+」が「02:54:44 」，「\w+\s+-- MARK --$」が「asao -- MARK --」とマッチします。

5行目で，ログの各行からホスト名，プログラム名，メッセージを抽出します。この正規表現では「(…)」が使われていますが，この丸かっこはくくった正規表現を一まとまりとして扱うためのものです。つまり「(\[\d+\])?」は，正規表現「\[\d+\]」の0回または1回の繰り返しを意味します。丸かっこを使うと，くくった正規表現にマッチした文字列を変数$1～$9を用いて参照できます。

例えば入力が**図7**のような行であった場合，5行目の正規表現中の「(\w+)」は，「asao」と，「([^:\[ ]+)」は「kernel」と，「(\[\d+\])?:?\s+」は「:」と，「(.＊)」は「Packet log: input DENY …」と，それぞれマッチしますが，このとき$1は「asao」，$2は「kernel」，$4は「Packet log: input DENY …」を、それぞれ参照することができます。従って，これらをそれぞれ変数$host，$name，$mesgへ代入します(9～11行目)。

なお，正規表現「[^:\[ ]」は，「[…]」でくくった，1文字にマッチする正規表現のいずれか，にマッチする正規表現ですが，角かっこ内の先頭に否定を意味する「^」があるため，「:」「[」「 」のいずれでもない1文字,という意味になります。前述した文字列の先頭を示す「^」とは意味が異なるので注意してください。

後は，変数$nameに代入されたプログラム名ごとに，変数$mesgに代入されたメッセージが重要なものか否かを判断して，重要でない行を読み飛ばす

表1　特定の1文字にマッチする正規表現
「α」～「\NNN」は，文字列中でも使用可能である。

正規表現	意味
.	改行以外の任意の1文字
\w	アルファベット(「_」を含む)
\W	非アルファベット
\s	空白
\S	非空白
\d	数字
\D	非数字
[…]	… のいずれか(…: 文字にマッチする正規表現の並び)
α	αと同一の文字(α:アルファベット1文字)
\☆	☆と同一の文字(☆:記号1文字)
\n	改行(0x0A)
\r	キャリッジ・リターン(0x0D)
\f	改ページ(0x0C)
\t	タブ(0x09)
\NNN	8進数NNNで文字コードを指定

表2　繰り返しを表す修飾子

修飾子	意味
	0回あるいは1回(「{0,1}」と同等)
＊	0回以上(「{0,}」と同等)
+	1回以上(「{1,}」と同等)
{n,m}	n回以上 m 回以下
{n,}	n回以上
{n}	n回(「{n,n}」と同等)

表3　文字列中の位置を指定する正規表現

正規表現	意味
^	文字列の先頭
$	文字列の末尾あるいは最後の改行文字の前
\b	単語の境界
\B	非単語境界

```
May  12  10:36:23  asao  kernel:  Packet  log:  input  DENY  eth0  PROTO=6
211.199.73.21:1087 210.145.125.175:111 L=60 S=0x00 I=51126 F=0x4000 T=44 SYN (#21)
```

図7　syslog が出力する行の例
実際には1行で出力される。

```
#!/usr/bin/perl
$ENV{'PATH'} = "/usr/bin:/usr/sbin:/bin:/sbin";
$Dev = "ttyS2";
$System = "ddip";
$Tel = "0570570011##4";
$User = "prin";
@PppOption = ("115200", "noipdefault", "defaultroute", "lock", "crtscts",
              "user", "$User", "remotename", "$System");
$| = 1;
open(MODEM, "+>/dev/$Dev");
vec($Rin, fileno(MODEM), 1) = 1;
&receive(0.1);
&send("ATD$Tel\r");
die "Fail to dial\n" unless &wait(120,"(CONNECT|BUSY|NO CARRIER)","CONNECT");
if (!fork) {
    close(STDIN);
    open(STDIN,"<&MODEM");
    close(STDOUT);
    open(STDOUT,">&MODEM");
    exec "pppd", @PppOption;
}
exit 0;

sub wait {
    my($sec, $exp, $suc) = @_;
    my($i);
    for ($i=0; $i < $sec*10; $i++) {
        &receive(0.1);
        return ($1 eq $suc || ! $suc) if /$exp/i;
    }
    0;
}

sub receive {
    my($timeout) = @_;
    my($rout);
    while ((select($rout=$Rin, undef, undef, $timeout))[0]) {
        if (vec($rout, fileno(MODEM), 1)) {
            exit 1 if sysread(MODEM, $_, 1024) <= 0;
            tr/\r\000\012\021\023\032/\n/d;
            $Buffer .= $_;
        }
    }
    $_ = $Buffer;
}

sub send {
    undef $Buffer;
    while ($_ = shift) {
        syswrite(MODEM, $_, length);
    }
}
```

図8　ppp接続を行うスクリプト

（つまり，next命令を実行する）スクリプトを追加していくだけです。

一例として，$nameが「syslogd」の場合（12～14行目）と「sshd」の場合（15～29行目）のスクリプトを示しました。sshdの場合，アクセス元ホストおよびユーザー名を調べて，特定の条件を満たせば「next」を実行します。「next」が実行されない場合は，30行目で標準出力へ出力することになります。ただし，同じメッセージが続く場合は最初の1行のみが出力されます。

ppp接続

ppp接続をするためのperlスクリプトを**図8**に示します。このスクリプトは，モデムにATコマンドを送ってプロバイダのアクセス・ポイントへダイアルし，接続完了後にpppdを実行してppp接続を行います。

perlにはデバッグのためのオプション「-d」があるので，効率的な開発が可能です。図8のスクリプトをppp.plというファイル名で保存した場合は，**図9**に示すように1行ずつスクリプトを表示しながら実行して動作を確認していくと良いでしょう。コマンドの一覧を表示するには，「h␣」を入力します。

スクリプトの4～6行目は，プロバイダのアクセス方法の設定です。アクセス・ポイントの電話番号およびユーザーIDの設定を行っています。パスワードの設定は，pppdの設定ファイルpap-secretsで行います（**図10**）。ここでは，DDIポケットのPHSを使ったインターネット接続サービス「PRIN」＊15の場合の設定を示します。設定情報を**表4**に示す。

7～8行目は，pppdのオプションです。詳しくはpppdのマニュアルを参照してください。9～14行目でアクセス・ポイントに対して電話をかけます。「MODEM」がモデム（例ではPHSカード）と入出力を行うためのファイル・ハンドルです。

無事アクセス・ポイントへ電話がつながる（モデムから「CONNECT」を受信する）と，16～19行目でファイル・ハンドルMODEMを標準入出力（STDIN，STDOUT）へリダイレクトして，pppdコマンドを実行します（20行目）。

24～32行目は，モデムから特定の文字列を受信するまで待つサブルーチンです。第1引数がタイムアウトまでの秒数，第2引数が期待する受信文字列パターンです。タイムアウトの場合0を返し，期待した文字列を受信したとき1を返します。ただし，第3引数が指定された場合，第2引数の「(…)」でくくったパターンにマッチする文字列が第3引数に一致する場合のみ，1を返します。

34～45行目は，モデムから文字列を受信するサブルーチンです。引数に与えた秒数の間，何も受信しなければそれまでに受信した文字列（変数$Bufferの内容）を返します。

47～52行目は，モデムに対して文字列を送信するサブルーチンです。

さて図8のスクリプトは，必要最小限の機能に絞って簡略化してあります。必要に応じて機能を追加してください＊16。例えば接続先プロバイダが複数ある場合，あるいはモデム等の通信手段が複数ある場合は，それらを簡単なオプションで切り替えられる方が便利でしょう。また，シリアル・ポートを他のプログラムからも使用する場合は，ロック・ファイルの制御が不可欠になります。

ここではスクリプトを機能拡張していく一例として，ロック・ファイル制御機能を追加する方法を説明します。まず，**図11**に示すサブルーチンを追加します。

```
asao:/home/sengoku % perl -d ppp.pl↵
Default die handler restored.

Loading DB routines from perl5db.pl version 1.07
Editor support available.

Enter h or `h h' for help, or `man perldebug' for more help.

main::(ppp.pl:2):       $ENV{'PATH'} = "/usr/bin:/usr/sbin:/bin:/sbin";
  DB<1> s↵
main::(ppp.pl:3):       $Dev = "ttyS2";
  DB<1> ↵
main::(ppp.pl:4):       $System = "ddip";
  DB<1> ↵
main::(ppp.pl:5):       $Tel = "0570570011##4";
  DB<1>
```

図9　perlスクリプトのデバッグ

```
prin    ddip    prin    -
```

図10　pppdの設定ファイル pap-secrets
1番目の「prin」がユーザーID，3番目の「prin」がパスワードである。

表4　PRIN設定情報
電話番号末尾の##4は，64k PIAFS（2.1版）でアクセスするための設定。2.1版はベスト・エフォート（best effort）型，2.0版はギャランティ（guarantee）型で，互換性が無いので注意が必要である。前者はDDIポケット，後者はNTTドコモのPHSカードで使われている。

電話番号	0570570011##4
ユーザID	prin
パスワード	prin

＊15　詳しくは，http://www.ddipocket.co.jp/data/prin.htmlを参照してください。手続き無しで，15円/分の全国均一の通話料金だけで64kbpsで接続できます。私は，接続先を3カ所に限定した「Two LINK DATA」PCカードを使っています。このPCカードは，あらかじめ登録した電話番号しかダイアルできないので，PRINなどの，電話番号が全国共通の接続サービス向きと言えます。
＊16　私が実際に使っている同種のスクリプトは400行以上あります。

```
 1  sub lock_check {
 2      if (-f "$LockPrefix$Dev") {
 3          open(LOCK,"$LockPrefix$Dev") || die "$!";
 4          if (read(LOCK,$pid,20) > 4) {
 5              $pid = $pid + 0;
 6          } else {
 7              $pid = unpack("L",$pid);
 8          }
 9          close(LOCK);
10           if ($pid != $$ && -d "/proc/$pid") {
11              die "Lock file exist !\n";
12          }
13      }
14  }
15
16  sub lock_make {
17      my $umask = umask(0);
18      open(LOCK,">$LockPrefix$Dev") || die "$!";
19      printf(LOCK "%10d\n",$$);
20      close(LOCK);
21      umask($umask);
22  }
```

図11　ロック・ファイル制御用サブルーチン
図8のスクリプトに追加する。

```
$LockPrefix = "/var/lock/LCK..";
```

図12　ロック・ファイル・プレフィックスの設定
図8のスクリプトの2行目に挿入する。

```
&lock_check;
&lock_make;
use FileHandle;
*MODEM = new FileHandle "/dev/$Dev", O_RDWR|O_NONBLOCK;
vec($Rin, fileno(MODEM), 1) = 1;
&receive(0.1);
&lock_check;
```

図13　ロック・ファイル制御用サブルーチンの呼び出し
図8のスクリプトの10～12行目をこの7行で置き換える。

```
    &lock_make;
```

図14　子プロセスのIDをロック・ファイルへ書き込む
図8のスクリプトの16行目に挿入する。

サブルーチンのlock_checkは，ロック・ファイルを調べ（2～13行目），他のプロセスが作ったロック・ファイルがあれば，終了します（11行目）。すなわち，ロック・ファイルにはそれを作ったプロセスのIDが書き込まれていますから＊17，書き込まれたIDを読み取って変数$pidへ代入し（4～8行目），そのIDが自身のプロセスIDでなく，かつそのプロセスが存在するか＊18調べます（10行目）。

サブルーチンlock_makeは，ロック・ファイルに自身のプロセスID（変数$$）を書き込みます（図11の19行目）。なお，変数$LockPrefixはロック・ファイルのパス名のプレフィックスです。ロック・ファイルのパス名が「/var/lock/LCK..デバイス名」という形式であれば，$LockPrefixに「/var/lock/LCK..」という値を設定しておきます。図8のスクリプトの2行目辺りに，**図12**に示す行を挿入しておくと良いでしょう。

次に，図8のスクリプトで，シリアル・ポートをオープンしている10～12行目を，**図13**に示す7行で置き換えます。すなわち，ロック・ファイルを調べ（lock_check），作成した（lock_make）後にシリアル・ポートを非ブロック・モード（O_NONBLOCK）＊19でオープンします。そして，ロック・ファイルを再度調べて正しくロック・ファイルが作成されているか確認します。なぜなら運悪く同時に他のプロセスがロック・ファイルを作成したかも知れないからです。もし他のプロセスのIDがロック・ファイルに書き込まれていた場合は，ファイル・ハンドルMODEMに対して送信してはいけません。図8のスクリプトでは，15行目で子プロセスを生成し，子プロセスとしてpppdを実行します。すなわちロック・ファイルに書き込むプロセスIDを，この子プロセスのIDに変更する必要があります。そこで，図8のスクリプトの16行目に，**図14**に示す行を挿入します。

＊17　プロセスIDをバイナリで書き込む方法と，文字列として書き込む方法があります。最近のLinuxでは後者が標準的なのではないかと思います。図11に示したサブルーチンは，バイナリが書き込まれている場合にも対応できます（7行目）。

＊18　Linuxでは，/procファイル・システムに，プロセスIDごとのディレクトリが現れます。従ってあるIDのプロセスが存在するか否かは，対応するディレクトリが存在するか調べる（図11の10行目）だけで済みます。このようなファイル・システムがサポートされていないOSでは，psコマンド等を実行してプロセスが存在するか調べる必要があるでしょう。

＊19　シリアル・ポートを複数のプログラムから利用する場合，非ブロック・モードでオープンする必要があります。詳しくは2001年3月号，pp.110-120の「実戦で学ぶ，一歩進んだサーバー構築術」を参照してください。