「情報処理」1年文I/IIクラス9-10　＃10

久野　靖*

1995.1.9

新年おめでとうございます。今年もよろしく。(あんまりよろしくしたくないかも知れませんが…)
さて、この科目も今回を入れて残すところあと4回(1/9、1/23、1/30、2/6)ということになります。毎回プログラミングが半分というのは変らないのですが、残り半分の内容として、これまでやってきた「〜の使い方」のようなものだけでなく、もう少し計算機の原理に関係のある内容もお話して行きたいと思います。今回の目標は次の通り。

- 数以外の「型」について学ぶ。
- 手続きについて復習する。
- お絵描きのアルゴリズムについて復習する。
- 「プロセスとプログラム」について学ぶ。

1　前々回の課題の解説

まず、配列を使った平均値と平均より大きいデータの個数のプログラム。図1にPADを再掲しておく。そして、Pascalは次の通り。

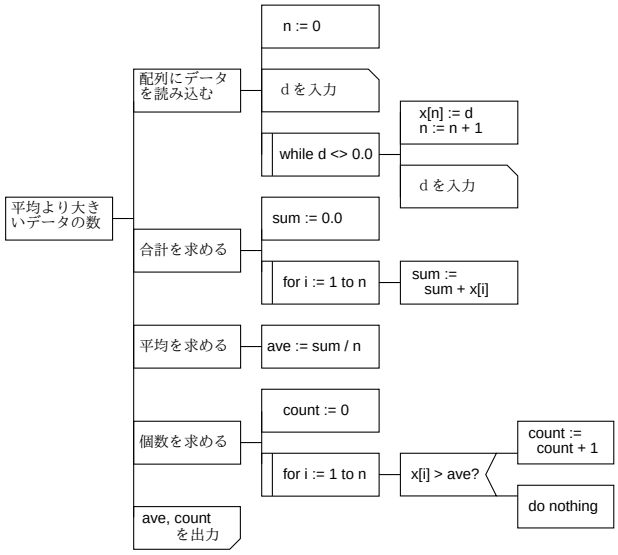

図 1: 平均より大きいデータの個数の PAD

*筑波大学大学院経営システム科学専攻

```
program sam12b(input, output);
var n, i, count: integer;
d, sum, ave: real;
x: array[1..200] of real;
begin
n := 0;
write('data = '); readln(d);
while d <> 0.0 do begin
n := n + 1; x[n] := d;
write('data = '); readln(d)
end;
sum := 0.0;
for i := 1 to n do sum := sum + x[i];
ave := sum / n;
count := 0;
for i := 1 to n do
if x[i] > ave then count := count + 1;
writeln('average = ', ave:8:3);
writeln('no. of data lager than average = ', count:1)
end.
```

では、「平均に一番近い」はどうしたらいいだろう? まず平均を求めるところまでは同じですね? その後、再度データを最初から調べて、「今のところ一番近いもの」を覚えればいいわけだ。なお、nearest の最初の値を x[1] にしているのはつまり、とりあえず 1 番目を一番近いと考え、その後もっと近いものがあれば順次置き換えて行くためである。

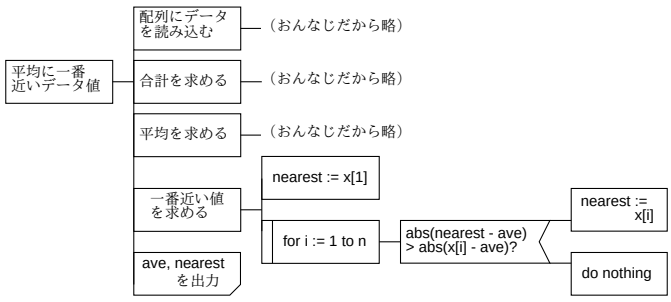

図 2: 平均に近いデータを求める PAD

```
program sam12b(input, output);
{ 途中略 }
ave := sum / n;
nearest := x[1];
for i := 1 to n do
if abs(nearest - ave) > abs(x[i] - ave) then nearest := x[i];
writeln('average = ', ave:8:3);
writeln('nearest = ', nearest:8:3)
end.
```

2　文字型と文字列

これまで Pascal で扱うデータは数値だけだった。それでも、整数を扱う integer と小数点つきの数を扱う real の 2 種類のデータ型 (値の種類) を使い分けていましたね? この辺でそろそろ、数値以外の「型」について学んでいただこう。

その手始めは文字型から始めよう。Pascal では文字型は char という名前で表す。char 型の変数は、1 つの文字を格納することができる「箱」である。文字には四則演算などはできないが、数と同様に read や write で読み書きしたり、:=で代入したり、ある文字と別の文字が等しいか、大小関係はどうかなど調べることができる。[1]

例えば 1 文字を読み込んで、それを 10 回打ち出すという簡単な例を示そう。

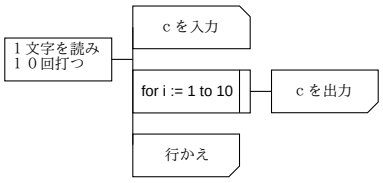

図 3: 1 文字読み 10 回打つ PAD

```
program sam15a(input, output);
var c: char;
i: integer;
begin
write('c = '); readln(c);
for i := 1 to 10 do write(c);
writeln
end.
```

実行例は次の通り。

```
% a.out
c = Baka
BBBBBBBBBB
%
```

「Baka」と 4 文字入れたのに「B」しか打ち出されていない…つまり、char 型は本当に 1 文字ぽっきりしか入らないわけである。それではさすがにつまらない…では 1 行ぶんの文字を読み込むにはどうしたらいいか?

実はその答えは既に知っている。つまり、同じ種類のデータが並んでいる場合には「配列」を使えば良い。例えば

```
var s: array[1..20] of char;
```

とすれば、変数 s には 20 文字までの文字を並べて入れることができる。そして、Pascal では read や write に文字の配列を指定して入出力できる。つまり、数値の配列と違っていちいちループを書かなくても 1 行まとめて読んだり書いたりできるのである。このため、「1 行読み込み、それを 10 回打つ」といった例題も次のように簡単である。

[1] 大小関係は文字符号のならび順による。具体的には $0 < 1 < \ldots < 9 < A < B < \ldots < Z < a < b < \ldots < z$ のようになる。より詳しくは、「man ascii」で調べること。

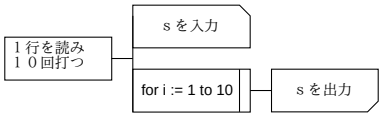

図 4: 1 行を読み 10 回打つ PAD

```
program sam15b(input, output);
var s: array[1..20] of char;
i: integer;
begin
write('s = '); readln(s);
for i := 1 to 10 do writeln(s)
end.
```

```
% a.out
s = This is a pen.
This is a pen.
This is a pen.
...
```

このように簡単であるが、配列である以上その i 番目を参照したり変更したりできる。ただしこの例では 20 文字までしか入らないのは当然である。

あと、入力する時 20 文字より少ない文字数を打ち込んだ場合には、配列の余った部分には空白が詰められる。だから実は上のプログラムは行末に余分の空白を打っているが、画面では見えないだけである。例えば「1 行読み込み、その中の空白を*に取り換える」という例をやってみればわかる。

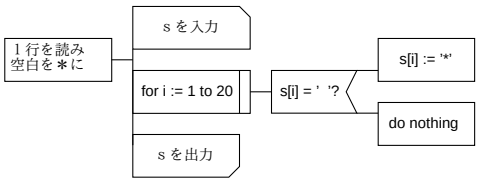

図 5: 1 行を読み空白を*にする PAD

```
program sam15c(input, output);
var s: array[1..20] of char;
i: integer;
begin
write('s = '); readln(s);
for i := 1 to 20 do
if s[i] = ' ' then s[i] := '*';
writeln(s)
end.
```

```
% a.out
s = This is a pen.
This*is*a*pen.******
```

まあ、あんまり気にしないで結構。どうしても余分な空白を出さないようにしたければどうしたらいいか?

1. 最後 (この場合は 20 文字目) から前に向かって見て、空白でない最初の文字を見つける。これが「本当の長さ」。
2. 書く時は 1 文字ずつ本当の長さまで書くか、または幅指定でこの本当の長さを指定する。つまり「writeln(s:l)」などとする。

では練習問題。

演習 1 次の PAD と Pascal プログラムを作り動かせ。1 行 20 文字までとしてよい。

a. 1 行読み込み、左右ひっくり返して打ち出す。(baka → akab のようになる。)
b. 1 行読み込み、各文字を 2 回ずつ打つ。(baka → bbaakkaa のようになる)。
c. 1 行読み込み、各文字を巡回させながら打ち出す。(baka → akab → kaba → abak のようになる。)

行末の空白はつけたままでもよい (取り除いて処理できればなおよい)。[2]

3 定数定義

よく見かけた間違いにこういうのがあった。

```
a: array[1..n] of real;
           ↑ここが変数
```

つまり、データの数は変数 n に入れるのだから、配列の大きさも n までいいだろうというわけである。しかし、Pascal ではそれはできない。なぜか? 配列の大きさぶんのメモリは、プログラムが実行を始める時に準備しなければならない。しかしその時には変数 n にはデータの個数がまだ入っていないわけである。しかし、これに代わる方法がある。それは、var よりも前に

```
const 名前 = 定数; …
```

という記述を入れることである。このとき、この名前を「定数名」という。この定数名は式の中で普通の定数の代わりに使うことができるし、また宣言の中で配列の範囲指定に使うこともできる。例えば、次のようなプログラムが作れる。

```
const xmax = 300;
      ymax = 200;
var   p: array[1..ymax, 1..xmax] of integer;
      ...
begin
  for i := 1 to ymax do
    for j := 1 to xmax do p[i, j] := 0;
  ...
```

なんだ、それでは 300 とか 200 と書くのと同じでは? と思われるかも知れない。確かにそうなのだが、しかし配列の大きさを 300 × 200 から変更したくなった時どうか? このようにしておけば、const の部分だけ置き換えればそれで済む。直接 300 とか 200 とか書いてしまうと、それを全部探して正しく置き換えるのは大変である。おわかりかな?

[2] ヒント: いずれも書く方は 1 文字ずつ write で書き出し、最後に引数なしの writeln を呼んで改行するとよい。

4　復習: 手続きについて

手続きとは「ある決まった手順を手続きとして定義すると、あとはそのような命令があるかのように使うことができる」というものだった。プログラムは、そのような新しい命令を使うととても簡単に書くことができるようになる。前回の例題を改良したもので示そう。どう改良するかというと、配列を 1 つから 3 つに増やして赤/緑/青の明るさ (0〜255 の値) をそれぞれに入れる。こうしておけば、いちいち「0 は白」などと対応を覚えておかなくてもいい。その代わり、これからは色を指定する時は 3 つの値の組を指定する。手続きを除いた部分は次の通り。

```
program sam16a(input, output);
const xmax = 300;
      ymax = 200;
var i, j: integer;
    red, green, blue: array[1..ymax, 1..xmax] of integer;
{ 手続き群の定義がここに入る }
begin
  clearscreen(200, 240, 255);
  for i := 1 to 8 do flower(30 + 30*i, 150+i+i, 20);
  writescreen
end.
```

この通り、画面をうす青にして、花を 8 個書き、画面を書き出してそれでおしまい、という簡単なプログラムである。画面をクリアする手続きは次の通り。

```
procedure clearscreen(r, g, b: integer);
begin
  for i := 1 to ymax do
    for j := 1 to xmax do begin
      red[i,j] := r; green[i,j] := g; blue[i,j] := b end
end;
```

ここで「r」「g」「b」は「パラメタ」であり、上のメインプログラムで「200」「240」「255」が渡されているから、それらの値がそれぞれ埋め込まれて実行される。

```
procedure writescreen;
begin
  writeln('P3 ', xmax:1, ' ', ymax:1, ' 255');
  for i := ymax downto 1 do
    for j := 1 to xmax do
      writeln(red[i,j]:1, ' ', green[i,j]:1, ' ', blue[i,j]:1)
end;
```

前回は出てこなかったが、パラメタのない手続きというのももちろん可能である。その場合は定義も呼び出しもかっこまで不要である。この手続きでは、まず「P3 幅 高さ 255」を書き、その後各点について赤、緑、青の値の組をどんどん書いて行く。

次に、点を打つ手続きを示す。これは、x と y の値が画面の範囲を出ていたら何もしない。こうしておけば、「BOT trap」とか「core dumped」とか言われる恐れはまずなくなる。(これらのエラーは、添字が用意した配列の範囲外を指してしまった場合に起きる。)

```
procedure point(x, y, r, g, b: integer);
begin
  if (x > 0) and (x <= xmax) and (y > 0) and (y <= ymax) then begin
    red[y,x] := r; green[y,x] := g; blue[y,x] := b end
end;
```

5 復習: 直線と円のアルゴリズム

さて、前回のアンケートで「数式がわかんない」というご意見を頂いたので、直線と円の書き方を復習してみよう。図 6 を参照されたい。点 (x_1, y_1) と点 (x_2, y_2) を結ぶ線分の上にある点 (x, y)

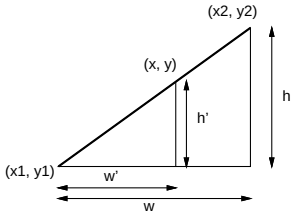

図 6: 線分の上の点の座標

の座標はどう表せるだろう? ただし、$\frac{w'}{w} = t$ とおく。すると、

$$x = x_1 + w' = x_1 + tw = x_1 + t(x_2 - x_1)$$

こうですね? そして、小さい三角形と大きい三角形は相似だから $\frac{h'}{h}$ も t に等しく、従って

$$y = y_1 + h' = y_1 + th = y_1 + t(y_2 - y_1)$$

こうなる。だから、x_1, y_1, x_2, y_2 が与えられているとすると、t を 0 から 1 までの間で十分細かく動かしながら対応する (x, y) を計算して、その付近に点を打って行けば、線分が描ける。どれくらい細かくか? 粗すぎると、点がとびとびになって点線になってしまう。細かすぎると、計算に時

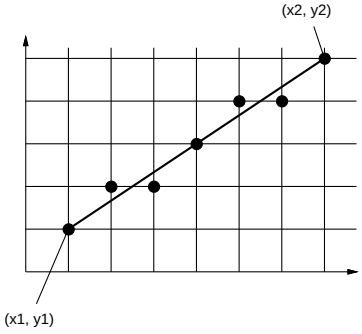

図 7: 点の集まりで直線を描くには

間が掛かる。要は x か y が 1 変化するごとに 1 個ずつ点を打てばいいのだから、w か h の大きい方を n として、n 個の点を打てばよいでしょう? というのがこの手続きである。

```
procedure line(x1, y1, x2, y2, r, g, b: integer);
var i, n: integer;
```

```
    dt: real;
begin
  n := abs(x2 - x1);
  if n < abs(y2 - y1) then n := abs(y2 - y1);
  dt := 1.0 / n;
  for i := 0 to n do
    point(round(x1 + (x2 - x1)*dt*i), round(y1 + (y2 - y1)*dt*i), r, g, b)
end;
```

なお、round(x) というのは値 x を四捨五入した整数値を返す。また、trunc(x) というのは値 x を切り捨てた整数値を返す。前にやりましたね? つまり、(x, y) を計算したとして、半端な位置には点は打てないのだから四捨五入した整数位置に直してそこに点を打つわけである。

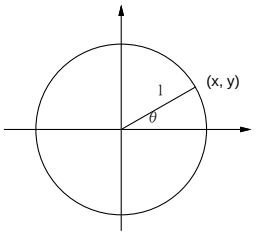

図 8: 円周上の点の座標

では次に、円を考えてみよう。図 8 のように、原点 $(0, 0)$ を中心とする半径 l の円がある時、x 軸方向を角度 0 として、角度 θ の位置の点 (x, y) の座標は

$$x = lcos\theta$$

$$y = lsin\theta$$

で表せますね? (まさか、これが分らない奴はいないだろうな…) で、(x_0, y_0) を中心とする半径 l の円の場合には、原点からその分だけ並行移動すればいいのだから

$$x = x_0 + lcos\theta$$

$$y = y_0 + lsin\theta$$

となるわけ。だから適当な間隔で θ を (この場合はラジアンだから)0〜2π の範囲で変化させると、円周が描ける。どれくらいの間隔が適当だろうか? それは角度が $d\theta$ 進むごとに円周上で 1 くらい距離が進めばいいわけだから、円周は $2\pi l$ なので

$$d\theta = \frac{2\pi}{2\pi l} = \frac{1}{l}$$

となるわけ。きざみ幅は 2π をこれで割ればよい。というわけで次の手続きができる。

```
procedure circle(x0, y0, l, r, g, b: integer);
var i, n: integer;
    dt: real;
begin
  dt := 1.0 / l; n := trunc(3.1416 * 2.0 / dt);
  for i := 0 to n do
    point(x0+round(l*cos(dt*i)), y0+round(l*sin(dt*i)), r, g, b)
end;
```

手続き flower はまあどうでもいいから (色の指定方法が変っただけ)、おまけにつけておく。

```
procedure flower(x0, y0, r: integer);
begin
  line(x0, y0, x0, y0 - r*6, 0, 255, 0);
  while r > 0 do begin
    circle(x0, y0, r, 255, 0, 0); r := trunc(r * 0.8)
  end
end;
```

これらをまとめると動くプログラムになる。なお、打ち込まなくても

```
cp  ~kuno/pascal/sam16a.p  sam16a.p
```

などのようにしてコピーして頂いてよい。

6 図形を描くアルゴリズム

さて、せっかくいろいろな色が使えるようになったのだから、塗りつぶした図形も描いてみよう。一番簡単なのは長方形ですね? 点 $(x0, y0)$ を左下隅として、巾 w、高さ h の長方形を指定した色で塗りつぶす手続きを書いてみよう。

```
procedure fillrectangle(x0, y0, w, h, r, g, b: integer);
var x, y: integer;
begin
  for x := x0 to x0 + w do
    for y := y0 to y0 + h do point(x, y, r, g, b)
end;
```

異常に簡単でしょう? これをさっきのに追加して、メインプログラムから呼んで見る。(資料がカラーでないので出力例は略。) プログラムのファイル名は `sam16b.p` なのでコピーしてよい。

```
begin
  clearscreen(255, 255, 255);
  fillrectangle(50, 20, 100, 100, 50, 200, 160);
  fillrectangle(100, 80, 150, 80, 150, 100, 0);
  writescreen
end.
```

さて、今度は中心 (x_0, y_0)、半径 l の塗りつぶした円をやってみる。塗りつぶすべき範囲は、点

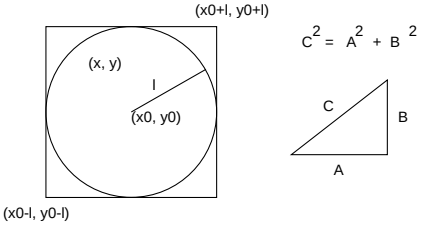

図 9: 円の内部の点の座標

$(x_0 - l, y_0 - l)$ を左下隅とした、幅高さとも $2l$ の正方形の中で、かつ点 (x_0, y_0) からの距離が l 以下の点、つまり

$$(x - x_0)^2 + (y - y_0)^2 \leq l^2$$

を満たすような点ですね。(まさかピタゴラスの公式は知ってますよね?)

```
procedure fillcircle(x0, y0, l, r, g, b: integer);
var x, y: integer;
begin
  for x := x0 - l to x0 + l do
    for y := y0 - l to y0 + l do
      if (x-x0)*(x-x0)+(y-y0)*(y-y0) <= l*l then point(x, y, r, g, b)
end;

begin
  clearscreen(255, 255, 255);
  fillcircle(50, 120, 40, 90, 200, 160);
  fillcircle(100, 80, 60, 250, 100, 0);
  writescreen
end.
```

ファイル名は `sam16c.p`。

演習 2 先の例題プログラムに次のような図形を描く手続きを追加してみよ。メインプログラムも対応して直し、動かしてみること。

a. 楕円。中心と x 方向と y 方向の半径を指定する。
b. 塗りつぶしたドーナツ型。中心と半径と「穴」の半径を指定する。
c. 7 色の虹。(手続きでなく、メインプログラムを直してもよい。ドーナツ型の虹でも許す。または、「まっすぐな」虹でも許す。)

7 プログラムとプロセス

7.1 マルチタスクとプロセス

皆様が使っている計算機システムは「Unix システム」と呼ばれているが、その Unix というのは「OS」(オペレーティングシステム) の 1 種類である。OS の機能には、前にやった「ファイルシステム」(ファイルやディレクトリを使えるようにする機能) などもあるが、今回はマルチタスク、つまり複数のプログラムを並行して走らせる機能について見てみることにする。そもそも、皆様は X 端末の前に座っているが、実際のプログラムは比較的少ない台数のサーバ計算機で動いている。ということは、1 台の計算機で沢山の人のプログラムを「同時に」動かしていることになる。どのようにしてそんなことが可能になるのだと思いますか?

まず、計算機の機能をごく簡単化して説明してみよう。メモリの上には命令の列 (プログラム) が置かれていて、CPU はプログラムカウンタが指している番地から順に命令を取り出しては実行して行く (図 10)。

そこで、複数のプログラムを並行して動かすには、それらをメモリに一緒に入れておく。一方、CPU に「時計」をつけておく (図 11)。そして、時計を動かした状態でまずはプログラム A の実行を始める。時計は一定時間たつと CPU に信号を送り、CPU は信号を受け取るとプログラム A

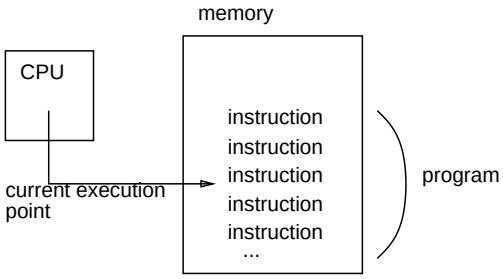

図 10: CPU による命令の実行

の実行を一時中断してプログラム B の実行に切り替わる。またしばらくするとプログラム C に、そして次は A に戻る。このようにして、複数のプログラムが実は「小刻みに切り替わりながら」実行されるのだが、CPU は非常に高速なのでユーザにとってはすべてのプログラムが同時に動いているようにしか見えない。

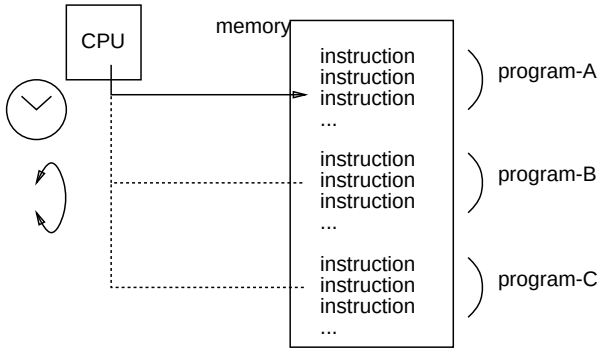

図 11: マルチタスク機能の実現

より厳密には、「プログラム」のうち 1 つは OS であり、時計から信号が来るとまず OS の実行に切り替わる。OS は各プログラムの使用時間の割り当てや優先順位を調べて次に実行すべきプログラムを選択し、それの実行を開始させる。だから OS によって各プログラムの CPU 割り当ては自由に制御できるわけである。

Unix 用語ではこの「動いている状態のプログラム」のことを「プロセス」と呼ぶ。(IBM 用語ではタスク、またアクティビティなどと呼ぶ文明もある。) たとえば 10 人の人が同時にある計算機で Mule を使っていれば、プログラムは 1 つだがプロセスは 10 あることになる。

ところで、計算機によっては CPU が複数ついているもの (マルチプロセサという) もある。しかしそのようなシステムでも、動かしたいプログラムの数 (プロセス数) は CPU の数よりずっと多いのが普通だから、上で述べたような小刻みな切り替わりがあることに変わりはない。

最後になったが、このように沢山プロセスが作れることはどういう利点があると思うか?

- 複数の端末をつないで、多人数で同時に使える
- 一人で複数の仕事を並行してこなせる
- 自分に代わって何かを監視するプログラムが動かせる
- 決まった時間になったらあることをする、というのができる
- あることをするために別のことをやめなくてもいい

もちろん、一つのプログラムに (聖徳太子みたいに) 沢山のことをやらせるのはがんばれば可能である。しかしそんなことで苦労するより、沢山プロセスを使ってそれぞれに簡単な仕事をするプログラムを走らせる方が作るのも管理するのも楽である。

7.2 ps — Unix でのプロセス観察

Unix では、「ps」というプログラムによって現在動いているプロセスを観察することができる。ps に与えるパラメタによって、表示するプロセスの範囲や詳しさを制御できる。

```
ps     --- 現在使っている端末 (窓) から起動した自分のプロセスの表示
ps x   --- 自分のプロセスすべての表示
ps ax  --- 他人のものも含めてすべてのプロセスを表示
ps lax --- 〃、ただしより詳しい表示
ps uax --- 〃、ただし CPU 使用量の多い順に、ユーザ名つきで表示
ps vax --- 〃、ただしメモリ使用量の多い順に表示
```

たとえばとあるサーバで ps ax を実行した結果を次に示す。

```
% ps ax
  PID TT STAT  TIME COMMAND
    0 ?  D     1:05 swapper
    1 ?  IW    0:02 /sbin/init -
    2 ?  D     0:02 pagedaemon
   61 ?  IW    0:01 portmap
   66 ?  IW    0:00 ypbind
   68 ?  IW    0:00 keyserv
   78 ?  I     0:02  (biod)
   79 ?  I     0:02  (biod)
   80 ?  I     0:02  (biod)
   81 ?  I     0:03  (biod)
   92 ?  S     0:09 syslogd
  104 ?  IW    0:00 sendmail: accepting connections
  109 ?  IW    0:00 rpc.statd
  111 ?  IW    0:00 rpc.lockd
  115 ?  S     4:28 automount
  118 ?  IW    0:00 /usr/etc/cesd
  119 ?  S     0:00 /usr/etc/ccv -f
  122 ?  S     0:00 /usr/etc/kkcv -f
  126 ?  D   167:32 update
  129 ?  IW    0:10 cron
  141 ?  IW    0:10 /usr/X11R5/bin/xdm -config /usr/local/etc/xdm/xdm-config
  144 ?  S     0:07 inetd
  148 ?  IW    0:00 /usr/lib/lpd
  187 ?  IW    5:14 /usr/local/Wnn4/bin/Wnn4/jserver
21277 ?  S     0:00 in.rlogind
  351 co IW    0:00 - cons8 console (getty)
21278 p0 S     0:00 -usr/local/bin/tcsh (tcsh)
21283 p0 R     0:00 ps ax
```

これらのうち、p0 などのように端末番号が記されているプロセスがおもにユーザが使っているもので、他のプロセスは大部分、システムのさまざまな作業を担っている。このように、Unix では複数のユーザが自分のために複数のプロセスを駆使しているのに加え、システム自体の運用のために多数のシステムプロセスが動いているのが普通である。

7.3 プロセスの新規生成

ではさっそく、プロセスを 1 つ作って見よう。

```
% ps x
   PID TT       S  TIME COMMAND
  9091 pts/11   O  0:00 ps x
  8823 pts/11   S  0:01 /usr/local/bin/tcsh
% xclock -a -u 1 &
[1] 9092
% ps x
   PID TT       S  TIME COMMAND
  9093 pts/11   O  0:00 ps x
  9092 pts/11   S  0:00 xclock -a -u 1
  8823 pts/11   S  0:01 /usr/local/bin/tcsh
%
```

「xclock…」を実行すると秒針つきの時計が画面に現われ、秒針が動いているのが見える。「ps x」で見ると確かに xclock というプロセスが増えているのが分かる。

実はふだんお世話になっている mule やコマンドの窓も同様にして作ることができる。

```
% mule &
[2] 9101
% kterm -e tcsh &
[3] 9102
% ps x
   PID TT       S  TIME COMMAND
  9092 pts/11   S  0:00 xclock -a -u 1
  8823 pts/11   S  0:02 /usr/local/bin/tcsh
  9102 pts/11   S  0:00 kterm -e tcsh
  9110 pts/11   O  0:00 ps x
```

ここで^X^C で mule を終了させたり、exit でコマンドの窓を終わらせたりすれば対応するプロセスも消滅する。このように、Unix ではこれまでの仕事と並行してなにかをさせるには、新しいプロセスを作ってそれにまかせるのが自然かつ簡単な方法である。

7.4 kill — プロセスの操作

ps の表示には必ず PID(プロセス ID) と呼ばれる番号が含まれる。これはプロセスの固有番号で、これを指定することによって自分のプロセスをいろいろに操作することができる。操作は kill 指令によってプロセスに各種のシグナルを送ることで行える。

kill -STOP プロセス ID	プロセスの実行を一時凍結する
kill -CONT プロセス ID	凍結したプロセスの実行を再開する
kill -TERM プロセス ID	プロセスに「終わってほしい」と信号する
kill -KILL プロセス ID	プロセスを強制終了させる。

なお、2 番目のパラメタを省略すると-TERM が送られる。また、標準設定ではコマンドを実行中に^C と^Z を押すとそのコマンドを実行しているプロセスに-TERM および-STOP シグナルがそれぞれ送られる。

7.5 プロセスの生成、コマンドインタープリタ

実はプロセスには「親子関係」がある。これは、どのプロセスがどのプロセスを生成したか、という関係のことである。例えば先の例で今度は「ps lx」を実行させてみる：

```
% ps lx
 F UID   PID PPID CP PRI NI    SZ RSS    WCHAN S TT       TIME COMMAND
 8  20 9092 8823 24  51 20   437 318 pollwait S pts/11  0:00 xclock -a -u
 8  20 8823 8822 80   0 20   265 223  ptm_dip S pts/11  0:02 /usr/local/bi
 8  20 9151 8823 14   0 20   206 173          O pts/11  0:00 ps lx
 8  20 9102 8823 30  20 20   494 400 pollwait S pts/11  0:00 kterm -e tcsh
```

この中の PID と PPID(Parent PID) を見てみると、あとから作った 3 つのプロセスの親は最初からある tcsh のプロセスになっている。言い替えれば tcsh のプロセスが ps その他のプロセスを生成している。

これは何を意味するだろう。実はあなたや私がキーボードから指令を打ち込むとそれは tcsh というプログラムによって読みとられる。tcsh はその文字のならびを見て、その内容に応じて求められているプログラムを走らせる (ということは、プロセスを生成する)。この様子を図**??**に示す。このように、利用者から指令を表す文字列を受けとってその内容に応じて内部の動作を起動するプログラムをコマンドインタープリタと呼ぶ。

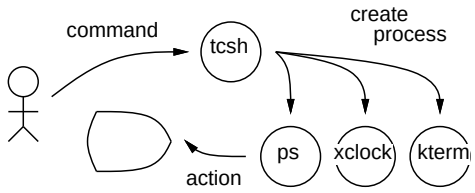

図 12: コマンドインタープリタ

コマンドインタープリタは Unix を使う上で欠かせない OS の一部ではあるが、一方でこれまでに見てきた、計算機が動き始める時まずメモリに読み込まれる OS の中核部分 (カーネル) とは違って、普通のユーザプログラムと同様にプロセスとして実行される。このように、メモリに常駐しなくてもよい部分はカーネルとは分けてプロセスとして実行させることでシステム全体の見通しをよくしているのも Unix の特徴である。(ps ax を実行したとき表示された多数のシステムプロセスも同様に Unix の機能の一部を担っているわけである。)

ところでさっきから毎回 ps を実行するごとに、その PID が違っていることにお気づきだろうか? つまりさっきの ps のプロセスは毎回実行が終れば消えてしまい、必要のつど新たに作られる。一方 tcsh そのものは同じままなわけである。この様子を図 13 に示す。つまり、tcsh はずっと動いたままで、利用者が指令を打ち込むたびに新しいプロセスが tcsh によって作られる。tcsh が終る

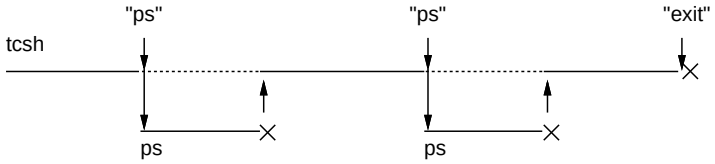

図 13: tcsh によるコマンドの発行と待ち合わせ

のは、利用者が exit という特別な指令を打ち込んだ時だけである。(ということは、exit というのは他の指令のように新しいプロセスとして実行されるのではなく tcsh 自身によって実行されることになる。このような「特別な」指令がいくつか存在する。)

ときに、図で点線のところは、tcsh が子供のプロセスの完了を待っていることを意味する。これは、普通利用者は一つの指令を打ち込んだらそれが終るのを待ってから次の指令を打ち込むだろうから、正しいあり方だといえる。でも指令がとても時間が掛かるようなものの場合には待っていたくないかも知れない。

そういうときは指令の最後に「&」をつけることで、「待たずにすぐ次の指令をやるよ」という指定ができる。さっき時計や mule などの窓を作るとき最後に&のついたコマンド行を使ったのはそういう意味だったのである。逆にいえば、&をつけるから新しいプロセスができるのではなく、いつでも新しいプロセスはできるのだが&をつけないとそのプロセスが終わるまで待つので次の指令を打つときにはもうそのプロセスはなくなっている、というだけのことだったのである。

演習 3 ps でプロセスを観察してみよ。具体的には次のことを行え。

- 自分のプロセスとしてはどんな名前のものがあったか?
- 新しく窓を作ったり、窓を終らせたりするとそれらはどう変化するか?
- 「xclock -a -u 1 &」で時計の窓を作り、プロセスが増えたことを確認せよ。次に、そのプロセスを kill で凍結/解凍してみて、凍結しておいた分だけ時間が遅れるかどうか調べよ。
- その他のプロセスも凍結したり殺したりしてみて、どのような影響が生じるか調べよ。

A　本日の課題 10A

本日の提出課題は「演習 1」から 1 つ (プログラムと実行例)、できた人は「演習 2」から 1 つ (xv の画面のみでよい)、そして「演習 3」の報告 (手書きでもプリントアウトでもよい) である。タイトルは「レポート 9A」とすること。アンケートは次の通り。

Q1. プログラムで文字を扱うことについてどう感じましたか?

Q2. プログラムによるお絵描きはできるようになりましたか? または、なりそうですか? どの辺が大変ですか?

Q3. プロセスを観察/操作してみてどうでしたか?

Q4. その他感想、要望などご自由にどうぞ。