計算機科学基礎'11 ＃2 – ソフトウェア構造/OS/コマンド

久野　靖*

2011.4.20

1　オペレーティングシステムとその役割り

1.1　アプリケーションソフトと基本ソフト

あなたがふだんソフトを使っているとき (たとえば Unix でも Windows でも Mac でもいいですが、ブラウザで WWW を見ているとします)、計算機の上で動いているのはそのプログラム (この場合はブラウザ) だけでしょうか? ブラウザを使っている最中でも、ちょっと別のプログラムを動かしたり、ブラウザの窓の位置を変更したりする時は、「ブラウザではない何か」を使って操作をしているでしょう? この「何か」の部分というのは、使うソフトを (たとえば表計算とかお絵描きとかに) 取り替えても同じまま…というか、そもそも各種のソフトを使う時の「土台」として常に存在している感じがするはずです。つまりソフトウェアには次の 2 種類があるわけです:

- **アプリケーションソフト**: ユーザが各々の仕事を実行するためのソフト。
- **基本ソフト**: アプリケーションソフトを使って仕事をする上で必要な手助け、ないし土台となるソフト。コンピュータを使うために必要となる仕事を行うソフト。

具体的には、どのような「手助け」が必要なのでしょうか? それはこれから徐々に見ていくことにして、ここではとりあえず基本ソフトを次のように分類しておきます。

- **オペレーティングシステム** (**OS**): アプリケーションが動く下ささえとなる機能を提供するひとまとまりのシステム。
- **ミドルウェア**: データベース管理システム (DBMS)、WWW サーバなど、アプリケーションの動く基盤となるが (この点は OS と同様)、OS とは分けて開発・提供されているもの。OS との区分はそれほど厳密でない。
- **言語処理系**: プログラムを作成するのに必要なソフトウェア。たとえば gcc(C コンパイラ) などがそう。
- **ユティリティ** ファイルの操作やデータ形式の変換など、(特定目的に特化した「アプリケーションソフト」とは対照的に) 汎用的な作業を手助けする。

分類する人の立場によっては言語処理系をユティリティに含めたり、ユティリティの中をさらに細かく分けるかも知れません。以下では、まず OS についてもう少し細かく検討し、その後でユティリティについて扱います。言語処理系は特に扱いませんが、C コンパイラ gcc はこれからも使う機会があると思います。ミドルウェアについてはデータベース、WWW など個々の話題の中で (別の回に) それぞれ取り上げます。

*経営システム科学専攻

1.2 OS の各種の役割り

上でオペレーティングシステムの役割りは「アプリケーションが動く下ささえ」と書きましたが、それは具体的にはどういうことでしょうか? もう少し具体的に考えてみましょう。

既に見たように、計算機のハードウェア命令というのは、メモリとレジスタの間でデータを転送したり、四則演算をしたりといったごく低いレベルの機能しか提供していません。ですから、多くのプログラムで必要とするような、キーボードを制御して文字列を読み込んだり、2 進表現のビット列を我々がふだん使っている 10 進表現に変換して画面に表示したりといった機能を実現するのには、かなり長い命令列 (プログラムの断片) を必要とします。

それを各アプリケーションを作る人が個別に用意するのでは労力の無駄ですし、そもそも普通のアプリケーションプログラマは入出力機器の制御方法など知らないのが普通です。たとえ知っていたとしても、個々のアプリケーションソフトで勝手に入出力機器を制御しはじめると、どれかのプログラムがキーボードの制御を握って離さなくなり他のプログラムではキー入力が使えない、など様々な問題が起きることでしょう。ですから、

- 多くのプログラムが必要とする標準的機能を一括して提供する

ことは OS の重要な役割りだと言えます (図 1)。

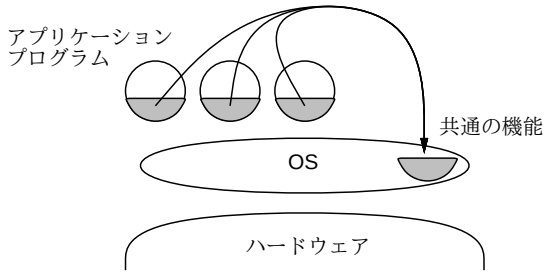

図 1: OS による標準的機能の提供

次に、現在の計算機システムでは複数のプログラムを並行して動かす**マルチタスク**機能が使われます。たとえば、ある窓で計算をさせながら、別の窓では待ち時間に WWW を見たり、といった具合です。このように、

- 複数のプログラムが並行して動作するのを管理する

ことも OS の役割りです (図 2)。

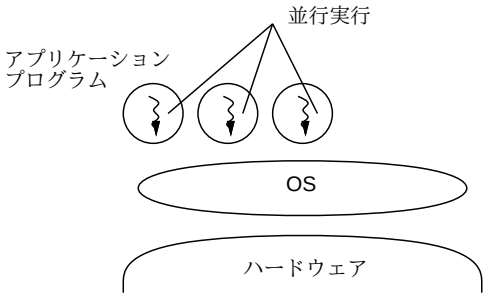

図 2: OS によるマルチタスク機能の提供

そうなってみると、あるプログラムを動かそうと思ったときいきなり計算機を止めてプログラムをメモリに書き込み始めるわけにはいきませんね。そんなことをすると現在進行中の別の仕事がめちゃくちゃになってしまいます (大体どうやって書き込むというのでしょうね?)。だから、

- ユーザが指定したプログラムを読み込んで実行開始させる

というのも OS の基本的な役割りなのです (図 3)。

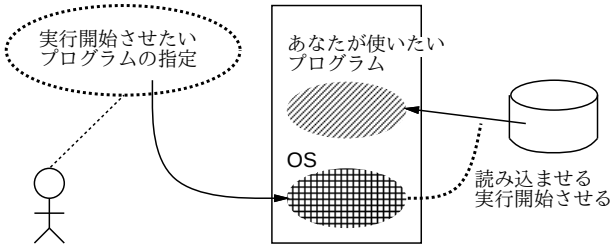

図 3: OS によるプログラムの実行開始

さて次に、そうやって並行して動いている複数のプログラムがたまたま同時に同じメモリ番地やディスク上の領域を使おうとしたら、やっぱり大混乱になるでしょうね? また、それらが一斉にプリンタに出力しようとして、混ざった出力が出てきても困るでしょうね? ですから、

- 計算機のメモリを各プログラムにうまく割り当てて調整する
- 入出力装置に対するアクセスを管理する

というのも OS の重要な仕事なわけです。

ところで見かたを変えると、計算機に備わっている入出力装置、メモリ、そして CPU などはすべて、数が限られた貴重な**資源**だと言えます。ですから、OS の機能のうち、マルチタスク機能、メモリ管理、入出力管理などは統一して

- 計算機内部の各種資源を管理する

ものと考えることができます。

言い替えれば、あなたが動いている計算機を目にする時、そこには常に OS が動いていて、ハードウェアと混然一体となってすべてを管理しているのです。そして、あなたやあなたのプログラムが計算機を使おうとする時、必要な資源のすべては OS が管理していて、(プログラムが)OS に頼むことによってはじめて、それらを利用して仕事ができるわけなのです (図 4)。

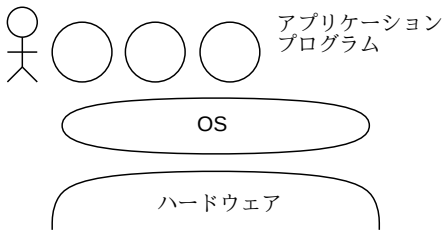

図 4: OS とユーザ、アプリケーションの関係

1.3 マルチタスクとプロセス

さて、OS には上に述べたように様々な機能が備わっていますが、まずはその中で一番目だつ機能であるマルチタスク、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ましょう。そもそも、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復習してみましょう。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きます (図 5)。

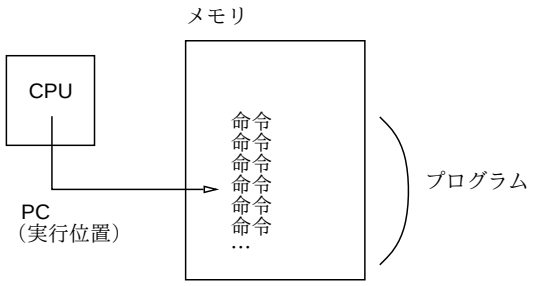

図 5: CPU による命令の実行

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きます。一方、CPU に**タイマー** (前章で述べたゲートを制御するクロックとは別の、もっと長い時間間隔で信号を送るような装置) をつけておきます (図 6)。そして、タイマーを動かした状態でまずはプログラム A の実行を始めます。

タイマーは一定時間たつと CPU に信号を送ります。CPU はタイマーから信号を受け取ると、プログラム A の実行を一時中断してプログラム B の実行に切り替わります。またしばらくすると実行はプログラム C に、そして次は A に戻ります。このようにすると、複数のプログラムが実は「小刻みに切り替わりながら」実行されますが、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わけです。

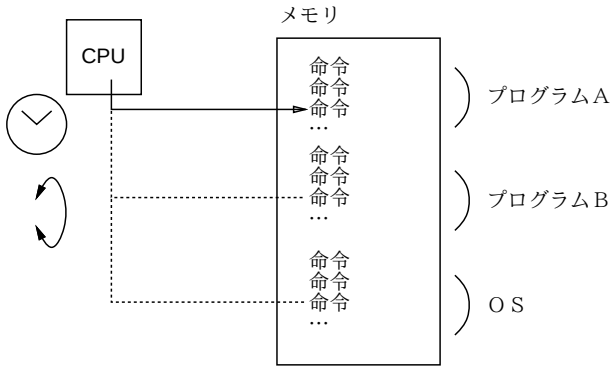

図 6: マルチタスク機能の実現

より厳密には、A〜C の「プログラム」のうち 1 つは OS で、タイマーから信号が来るとまず OS の実行に切り替わります。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、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ます。ですから、OS は各プログラムへの CPU 割り当てを自由に制御できるのです。

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** (process) と呼びます。たとえば 10 人の人が同時に 1 つのマシンに接続してそこで emacs エディタを使っていれば、「プログラムは 1 つ」ですが、「プロセスは 10」あることになります。

前回見たように、CPU を複数搭載したシステム (**マルチプロセサ**や**マルチコア**) も今日では珍しくありません。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ですから、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あり

ません。[1]

最後になりましたが、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でしょうか? たとえば次のような答えがあるかと思います。

- 複数の端末やネットワークを介して、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、がんばれば可能でしょう。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なのです。

つまり、CPU は 1 個、メモリは 1 枚しかなくて、その上で直接プログラムを走らせるなら 1 個だけしか走らせられないけれど、その上にプロセスという「プログラムが走るいれもの」を作り出すことで、多数のプログラムを並行して走らせて色々な作業をこなさせることができるわけです。

このように、「実際にはないけれどソフトウェア (OS など) の働きによって役に立つものを作り出す」ことを**仮想化**と言い、計算機システムでは非常に多く使われる考え方です。そして、プロセスは OS の働きによって作り出されている、「仮想化された CPU とメモリ」なわけです。[2]

1.4 ps — Unix でのプロセス観察

Unix では、**ps**(process status — プロセス状態 — の略) 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ます。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ます。[3]

- `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になる
- `ps ax` — 他人のものも含めた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

なお、我々のサイトではサーバマシンは安全のため「自分のものでないプロセスは観察できない」ように設定してあります。各自が直接使うマシンについてはこのような制限はありません。

たとえばあるマシンで `ps ax` を実行した結果を次に示します。

```
  PID  TT  STAT      TIME COMMAND
    0  ??  WLs    0:00.01 [swapper]
    1  ??  ILs    0:00.03 /sbin/init --
    2  ??  DL     0:20.55 [g_event]
  (途中省略)
   43  ??  DL     0:15.65 [schedcpu]
```

[1] ところで、上で説明してきたような「小刻みな切り替え」の間隔はどれくらいだと思いますか? Unix のようなシステムでは、1 ミリ秒 (回数でいうと 1 秒間に 1000 回) 程度ですが、これで十分「同時に動いている」感じになります。

[2] OS のうちでプロセスを作り出す機能の部分を**プロセス管理**と呼びます。

[3] Unix システムの系統によって指定方法や指定できる内容が違っています。ここで説明しているパラメタ指定は FreeBSD のものです。

```
   97  ??  DL     0:00.04 [md0]
  159  ??  Is     0:00.00 adjkerntz -i
  467  ??  Is     0:00.00 /sbin/devd
  573  ??  Is     0:00.54 /usr/sbin/syslogd -s
  594  ??  Ss     0:00.34 /usr/sbin/rpcbind
  626  ??  Ss     0:06.10 /usr/sbin/amd -p -a /.amd_mnt -l
  690  ??  Is     0:00.01 /usr/sbin/lpd
  705  ??  Ss     0:10.32 /usr/sbin/ntpd -c /etc/ntp.conf
  727  ??  Ss     0:06.64 sendmail: accepting connections
  731  ??  Is     0:00.12 sendmail: Queue runner@00:30:00
  737  ??  Is     0:00.72 /usr/sbin/cron -s
  767  ??  Is     0:00.01 /usr/sbin/inetd -wW -C 60
  785  v0  Is     0:00.02 login [pam] (login)
  815  v0  I      0:00.04 /bin/tcsh
13557  v0  I+     0:00.01 /usr/local/Xorg-7.2/bin/xinit.bin
13559  v0  S     10:16.23 /usr/local/XF86-4.6.0/bin/XFree86 :0
13560  v0  S      0:00.43 kterm -C -n console -T console
13569  v0  I      0:01.11 twm
13579  v0  S      0:00.87 xbiff -geom 101x101-204+0
  786  v1  Is+    0:00.00 /usr/libexec/getty Pc ttyv1
  787  v2  Is+    0:00.00 /usr/libexec/getty Pc ttyv2
  677 con- I      0:01.19 /lbin/ewhod utogw
13593  p0  Ss     0:00.12 tcsh
17552  p0  R+     0:00.00 ps ax
```

これらのうち、TT 欄に「p0」などのように**端末番号**が記されているプロセスは、ユーザが直接使っているものです。そして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ます。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なのです。

1.5 プロセスの新規生成

ではさっそく、プロセスを 1 つ作って見ましょう。

```
% ps x
  PID  TT  STAT      TIME COMMAND
57468  p0  Ss     0:00.33 bash
59157  p0  R+     0:00.00 ps x
% xclock -analog -update 1 &
[1] 59392
% ps x
  PID  TT  STAT      TIME COMMAND
57468  p0  Ss     0:00.53 bash
59392  p0  S      0:00.16 xclock -analog -update 1
59394  p0  R+     0:00.00 ps x
%
```

「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ます。2 回目の「ps」の出力を見ると確かに **xclock** というプロセスが増えているのが分かります。xclock に限らず、エディタ **emacs** や**コマンド窓**のプログラム **kterm** も同様にして動かすことができます。

```
% emacs &
```

```
[2] 59409
% kterm &
[3] 59411
% ps x
  PID  TT  STAT      TIME COMMAND
57468  p0  Ss     0:00.53 bash
59392  p0  S      0:00.16 xclock -analog -update 1
59409  p0  I      0:01.25 emacs
59446  p0  R+     0:00.01 ps x
59411  p2  Is+    0:00.09 bash
```

ここで emacs を終了させたり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します。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なのです。[4]

1.6 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ます。これはプロセスの固有番号であり、これを指定して **kill** コマンドによってプロセスに各種のシグナルを送ることで、自分のプロセスをいろいろに操作できます。

- `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 が送られます。また標準設定では、コマンドを実行中に ^C を押すとそのコマンドを実行しているプロセスに-TERM 相当のシグナルが、^Z を押すと-STOP 相当のシグナルが、それぞれ送られます。[5]

1.7 プロセスの生成、コマンドインタプリタ

実はプロセスには「親子関係」があります。これはつまり、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す。例えば先の例で今度は「`ps lx`」を実行させてみます:

```
% ps lx
UID   PID  PPID CPU PRI NI  VSZ  RSS WCHAN  STAT TT     TIME COMMAND
 21 57468 57459   1  10  0 1736 1212 wait   Ss   p0  0:00.53 bash
 21 59392 57468   0   2  0 2756 1556 select S    p0  0:00.11 xclock -ana
 21 59409 57468   0   2  0 7244 5184 select S    p0  0:01.24 emacs
 21 59422 57468   0  28  0  388  216 -      R+   p0  0:00.00 ps lx
 21 59411 59410  22   3  0 1728 1248 ttyin  Ss+  p2  0:00.09 bash
```

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 bash のプロセスになっています。言い替えれば bash のプロセスが ps その他のプロセスを生成しているわけです。

---

[4] kterm のプロセスが表示されていないな、と思いましたか? kterm はその仕事の都合上、root という特別なユーザで実行されるようになっているので、「自分のプロセスを表示」させても表示されません。全部のプロセスを表示させればちゃんと見えます。

[5] 「相当の」というのは、いちおう区別のためにシグナル番号は違えてあるけれど機能的には同じという意味です。

これは何を意味するのでしょう? 実はあなたや私がキーボードからコマンドを打ち込むと、それは bash というプログラムによって読みとられるのです。bash 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実行開始させる (具体的には、OS に依頼してプロセスを生成する) わけです。この様子を図 7 に示します。このように、利用者からコマンド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、一般に**コマンドインタプリタ**と呼びます。

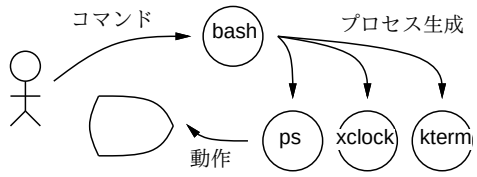

図 7: コマンドインタプリタ

コマンドインタプリタは Unix を使う上で欠かせない基本ソフトの一部ですが、これまでに見てきた 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のプロセスとして実行されます。こうしておけば、コマンドインタプリタを改良するなどして置き換える場合も、OS 全体ではなくそのプログラム部分だけ取り換えれば済みますし、ユーザの好みに応じて複数のコマンドインタプリタを提供することもできます (実際そうなっています — あとで説明しましょう)。そして、コマンドインタプリタに限らず、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す。

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でしょうか? つまり、ps のプロセスは 1 回表示を行うだけで直ちに消えてしまい、必要のつど新たに作られるわけです。一方、bash そのものは同じままです。この様子を図 8 に示しました。つまり、bash はずっと動いたままですが、コマンドの方は利用者がコマンドを打ち込むたびにそのコマンドのプログラムを実行する新しいプロセスが bash によって作られるのです。

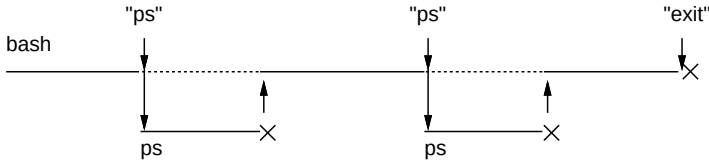

図 8: bash によるコマンドの発行と待ち合わせ

bash が終るのは、利用者が exit という特別なコマンドを打ち込んだ時だけです。(ということは、exit というのは他のコマンドのように新しいプロセスとして実行されるのではなく、bash 自身によって実行されることになります。このような「特別な」コマンドがいくつか存在します。)

ときに、図で点線のところは、bash が子供のプロセスの完了を待っていることを意味します。普通利用者は一つのコマンドを打ち込んだらそれが終るのを待ってから次のコマンドを打ち込むでしょうから、これが期待される動作だといえます。でも、コマンド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ません。

そういうときは指令の最後に「&」をつけることで、「待たずにすぐ次のコマンドを打ち込みたいよ」という指定ができます。さっき xclock や emacs 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わけです。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ですが、&をつけないとそのプロセスが終わるまで待つので、次のコマンド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

たのです。

2　シェルとその機能

2.1　コマンド入力のユーザインタフェース

ここまで繰り返し見てきたように、Unix では

- コマンドを打ち込むと、それに応じてプログラムが実行される
- どのような動作が行なわれるかは、起動するプログラムとそのプログラムに与えるパラメタによって決まる

という形でユーザがシステムに指示を与えています。このような方式を**コマンド行インタフェース**と呼びます。そして、コマンドを解釈してそれに対応する動作を起動してくれるプログラムを**コマンドインタプリタ**と呼びます (図 9)。Unix では伝統的に、コマンドインタプリタのことを**シェル**と呼びます。[6]

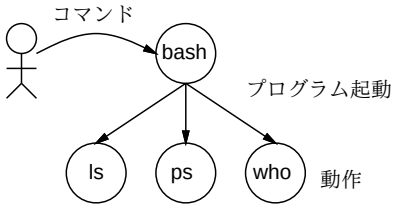

図 9: コマンドインタプリタ

このような方式について「コマンドを覚えるのが大変だ」「コマンドを打ち込むのが負担だ」「だから Unix は使いづらい」と思われている人が多いかも知れません。でも「使いやすい」とは、厳密にはどういう意味なのでしょうか?

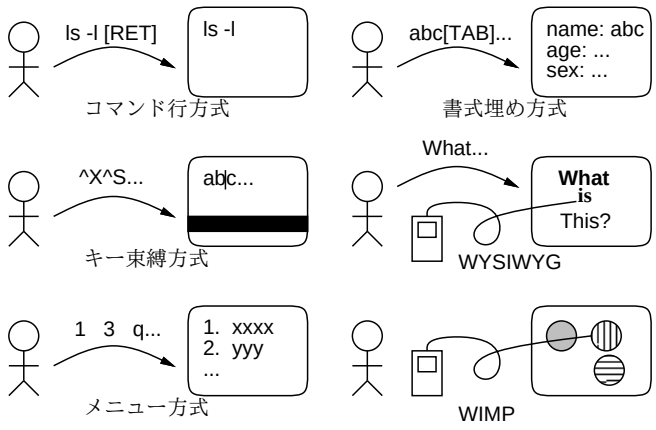

図 10: コマンド入力の各種方式

計算機の世界では、利用者と計算機の間でやりとりをする方式や機構のことを総括的に**ユーザインタフェース**と呼んでいます。PC や携帯電話、PDA などを色々見くらべるとわかるように、世の中には様々なユーザインタフェースが存在しています。その代表的な方式を図 10 に示しました。

[6]利用者を囲って守ってくれる「貝殻 (shell)」になぞらえてそう呼ぶようになったそうです。

- コマンド行方式: Unix のシェルのように、コマンドをキーボードから文字列として打ち込み、最後に [RET] などを押すと実行される。
- キー束縛方式: Emacs のように、制御キーを押すと対応する動作が実行されるというもの。テキストエディタなどテキスト入力とコマンドが混在する場合に多く使われる。
- 書式埋め方式: 画面に「記入欄のある書式」が表示され、矢印キーなどで欄を移動して欄に適切な文字列を打ち込み、完成したら [RET] キーなどで実行する。予約システムなどで多く見られる。
- メニュー方式: メニューが常時画面に表示されているか、または「メニューキー」のようなものを押すと表示され、矢印キー、番号キーなどで項目を選んで選択すると動作が実行される。今日の携帯電話のインタフェースはこれが大多数である。
- 直接操作方式: 操作したい対象を画面上で (マウスなどによって) 直接つかんだり移動したりして操作する。ワープロやお絵描きツールなど、「画面で見えるもの」が「最終生成物」に対応させられる場合に多く使われ、その場合は「What Yous See Is What You Get」(**WYSIWYG**) と呼ばれる。
- アイコン方式: 操作したい対象を小さな絵など (アイコン) で表し、それをマウスなどで選択し、移動したり重ねたりメニューを出したりして操作する。「Windows, Icons, Menus, Pointing」(**WIMP**) インタフェースとも呼ばれる。

多くの人が「使いやすい」と感じるのはメニュー方式、直接操作、アイコン方式といったあたりだと思われます。これらはいずれも「マウスなどで対象を直接指示するのでわかりやすい」「メニューで可能な選択肢が向こうから示されるので覚えなくてすむ」という利点を持つので、初めての人にも取りつきやすいからです。

逆に、古くから Unix で主に使われて来たのはコマンド行方式やキー束縛方式で、これらはどちらも何を打ち込むとどんな動作が起こるかを覚えていないと使えません。だから初心者に敷居が高いのは当然のことです。[7]

ではどうして「使いにくい」コマンド行方式やキー束縛方式が絶滅しないのでしょうか? それは、初心者には敷居が高い代わりに、習熟するととても高速に操作でき、柔軟性も高いという利点があるからです。逆に言えば、コマンドなどを覚えてしまった人に取ってはメニューや直接操作やアイコンは「操作に時間が掛かっていらいらして使いづらい」のです。[8]

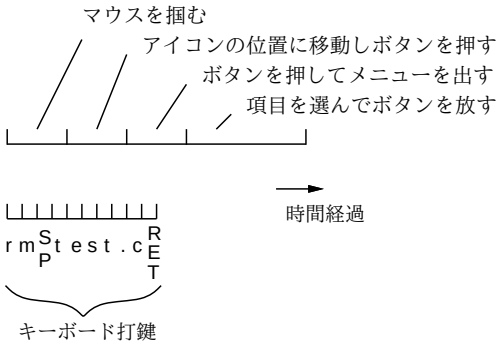

図 11: アイコン+メニュー操作とコマンド入力の時間比較

操作に時間が掛かるというのはどういう意味でしょうか? もっと具体的に見てみましょう。

[7] Unix でも GUI のツールを動かしてマウス操作だけで大抵のことが行えるようにもできます。このあたりの話題は次の章で取り上げます。

[8] 書式埋め方式はこれらの中間で、「どんなものを」打ち込むべきか、またその標準値はいくつか、といった情報は書式によって示せますが、具体的に「何を」打ち込むかは覚える必要があります。

- 直接操作方式でもアイコン方式でも、操作の対象をマウスで選ぶのには一定 (1 秒前後) の時間が掛かる。また 1 画面に入れておける対象の数は限られていて、画面を切り替える場合にはさらに掛かる。
- メニューから項目を選択するのにも一定の時間が掛かる。このためメニューの一部をキー束縛で選べるように工夫したり、アイコン方式でよく使う操作はメニューを経ないで選べるようにしたりする。しかしそうやって「近道」を用意できる操作の数は限られている。
- メニュー方式では、あらかじめメニューに入れてある動作しか指定できない。また、1 つの (ないし 1 画面ぶんの) メニューに入れられる動作の数には限りがある。アイコン方式でも、1 画面に入れておけるアイコン数は限られている。
- メニュー方式でもアイコン方式でも、動作に対するオプション (追加指定) のようなものは別のやり方で与えなければならない。

これに対し、キーボードの打鍵は 200msec 程度しか時間を要さないので、キー束縛方式では 1 つの動作がとても高速に指定できます。コマンド行方式では「文字をいろいろ打ち込む」ことで様々なオプションを自由に指定できます。

たとえば、図 11 にあるように「アイコンを選んで、メニューを出して、コマンドを選択する」という動作はどうやっても 5 秒〜10 秒くらいの時間が掛かります。一方、コマンドを打ち込む場合はコマンドの文字数にもよりますが、2〜4 秒くらいで済むことが多いのです (少なくとも慣れている人なら)。となると、「使いやすさ」を「短い時間で操作できる」というふうに定義したとすれば、慣れている人にはコマンド方式の方が「使いやすい」ことになるわけです。

もちろん、ここで「だから皆がコマンド行方式やキー束縛方式を使うべきだ」というつもりはありません。ただ、せっかく Unix を学ぶのですから、このようなインタフェースを体験してみて、その強力さや柔軟性を味わってみて頂きたいと思います。そうしておけば、さまざまな場面でユーザインタフェースについて考えるときに「とりあえず Windows しか知らないから Windows みたいにしておこう」という人よりはうまく考えられるはずです。[9]

では次節以降で、Unix のシェルを題材としてそのさまざまな機能を見て行きましょう。なお、既に述べたように Unix では複数のシェルが利用できるのが普通です。これらは大きく次の 2 つに分けられます。

- **Bourne シェル系**: Unix バージョン 7 で作られた「Bourne シェル」(Bourne は作者の名前) から派生し、これと上位互換のもの
- **C シェル系**: バークレー版 Unix と呼ばれる広く普及した系列の Unix で導入された「C シェル」と呼ばれるシェルに上位互換のもの

本資料では Bourne シェル系のシェルの 1 つである **bash** を中心に取り上げています。

2.2 コマンドとは?

まず最初に、コマンドとは一体何でしょう? `ps` コマンドの表示を見ると `ps` というプログラムがプロセスとして動いているのが観察できました。また `emacs`、`kterm`、`xcolck` などもそのまま同名のプログラムでした。つまり、シェルではコマンドとはプログラムの名前に他ならないわけです。より正確に言えば、プログラムの実行形式が格納されているファイルの名前がコマンド名でもある、ということになります。

[9]現に携帯電話ではマウスがありませんし画面も小さいので、アイコンのようなものはあまり使われず、メニューをボタンで選択するインタフェースが主流です。

たとえば gcc で C のソースプログラムをコンパイルすると、a.out という実行形式ファイルになりました。そしてその実行形式を動かすには、ファイルの名前 a.out をコマンドとして打ち込みましたね。では、a.out ではなく別の名前だったらどうでしょうか?

```
% cat hello.c          ←ソースを見る
main() { printf("Hello.\n"); }
% gcc hello.c          ←コンパイルする
% ls
a.out   hello.c       ←実行形式ファイル: a.out
% a.out               ←ファイル名を打つと実行
Hello.
% mv a.out hello       ←ファイル名を変更してみる
% ls
hello   hello.c        ← hello という名前に変更
% hello               ←ファイル名を打つと実行
Hello.
%
```

つまり、ファイルの名前を取り換えれば、その取り換えた名前が新しいコマンドの名前になるわけです。

OS の目的を「プログラムを実行させること」と考えるなら、「実行したいファイルの名前を言うことがすなわちコマンド」というシェルの方針は、これ以上ないくらい単純明快だと言えるでしょう。

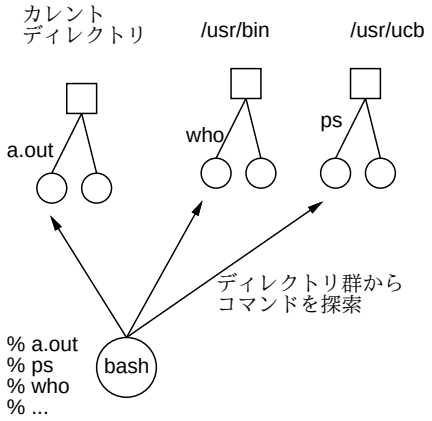

図 12: シェルによるコマンド探索

しかしそれにしても、自分は ls とか ps とかいうファイルは持っていないけど、と思われたかも知れません。もちろん、こういう共通のコマンドの実行形式ファイルをそれぞれの人が持つのではディスクの無駄ですから、共通のコマンドに対応する実行ファイルは共通の場所 (/bin、/usr/bin、/usr/local/bin など) においてあり、シェルはそれらのディレクトリを順番に探して実行形式ファイルを見つけるようになっています。具体的にどこにあるかを知りたければ type というコマンドを使います:

- type コマンド名 — コマンドの種別やありかを表示

```
% type ls
ls is /bin/ls
%
```

つまり ls というコマンドは/bin というディレクトリにあると分かったわけです。
なお、この「順番に探す」という動作はコマンドに「/」が含まれている場合には行われません。「/」が入っている場合は、ファイルを直接指定しているものとして扱われます。

2.3 コマンドの組み合わせとリダイレクション

シェルの特徴の 1 つとして、1 つのコマンド行で複数のプログラムの実行を指示したり、さらにそれらのプログラム群の入出力関係や実行順序を制御できることが挙げられます。
まず、1 つのコマンドに対してその標準入力や標準出力の接続先を切り替えるのには、前章で説明したようにリダイレクションを用います。

コマンド <入力ファイル
コマンド >出力ファイル

両方同時に指定してももちろん構いません。なお、「>」では既に出力ファイルに内容が入っている場合には一担からっぽにされますが、

コマンド >>出力ファイル

という形のリダイレクションなら既にある内容の後ろに追加されます。
ここまでは入出力をファイルに切り替えるという話だけでしたが、さらにシェルでは、あるコマンドの出力を別のコマンドの入力に接続することができます。これを**パイプライン**と呼んでいます (CPU 命令のパイプライン実行とは別の意味ですので注意)。

コマンド 1 | コマンド 2 | ... | コマンド n

この記法については、あとで沢山使うことになります。

2.4 シェル変数 option

多くのプログラミング言語では、値を蓄えておくために**変数**を使用します。シェルは後で出て来るように、小規模ながらプログラミング言語としての機能も持っているため、やはり変数の機能を持っています。変数に値を設定するのには (これまた他の言語と同様に)「=」を用います。

```
% a=0123456789
```

bash など Boune シェル系のシェルでは「=」の前後に空白を入れてはいけないので注意してください。[10]
一方、変数の内容を参照する場合には変数名の前に「$」をつけることになっています。

```
% cp t.c $a$a$a
% ls
01234567890123456789012345678 9  t.c
%
```

これを見ると、$a のところが変数 a に格納した値 0123456789 で置き代わっていることがわかります。なお、変数の値を調べるにはいちいちファイルを作らなくても引数をそのまま標準出力に打ち出す指令 echo を使えばよいのです。

[10] C シェル系では「set a = 0123456789」のように set というコマンドを使います。

```
% echo $a
0123456789
%
```

シェル変数に値を入れておけるのは分かりましたが、これは何の役に立つのでしょうか? 例えば次のようなことが考えられます:

- 現在位置付近にないファイルやディレクトリをくり返し参照したいとき、そのパス名を覚えさせておく
- 長いコマンド名や、複雑なコマンド引数などを覚えさせておく

これに加えて、すぐには思い付かないかも知れませんが、シェル変数は次の用途にも使われます。

- シェルと利用者の間での情報の受け渡しに用いる
- 利用者が動かすプログラムへの情報伝達に用いる

以下では、これらの使い方について説明していきます。

2.5 組み込みシェル変数 option

シェル変数のうちのいくつかは、利用者が値を設定しなくても最初から値が設定されています。具体的には次のものがそうです:

- $USER — 利用者のユーザ名
- $UID — 利用者のユーザ ID
- $HOME — 利用者のホームディレクトリ
- $SHELL — 利用者が使っているシェル
- $PATH — コマンドを探すディレクトリのリスト
- $TERM — 端末の種類
- $PS1 — プロンプト文字列
- $PS2 — 〃 (継続行用)

そして、これらのいくつかは値を設定することによってシェルの動作を好みに合わせて調整できます。たとえば、プロンプトにカレントディレクトリやホスト名などを表示させることもできます:

```
% PS1='>>> '
>>> PS1='\w> '
~/text/tex/ecs94/sample> PS1='\u@\h '
kuno@smb PS1='% '
%
```

PS1 を設定したとたんに、シェルのプロンプトはその値に変化してしまうことに注意。なお、bash ではプロンプト文字列の中にカレントディレクトリなどを表示させるため,、つぎのような制御列を書くことができます:

\t	現在時刻
\w	カレントディレクトリのパス名
\W	カレントディレクトリ名
\u	ユーザ名
\h	ホスト名

もう 1 つ重要なシェル変数として **PATH** があります:

```
% echo $PATH
.:/usr/local/X11R6/bin:/usr/local/bin:/usr/ucb:/usr/bin:/bin
%
```

すなわち、PATH にはコマンドの実行形式を集めたディレクトリのリストを「:」で区切ってならべたものが入っています。ここに自分のサブディレクトリを追加すると、そのサブディレクトリに入れた実行形式ファイルは他のコマンドと同様に使えるようになります:

```
% PATH=$HOME/bin:$PATH       ← PATH に自分のディレクトリを追加
% mkdir $HOME/bin            ←そのディレクトリを作る
% mv a.out $HOME/bin/hello   ←そこに hello というコマンドを入れる
% hello
Hello.
%
```

これ以外にも、シェルの動作を制御する組み込みシェル変数がいくつかありますが、これらについてはそれぞれの機能の説明のところで述べます。

2.6 環境変数 option

シェル変数を拡張して、シェル以外にそのシェルから起動したプロセスにも情報を渡せるようにしたものを**環境変数**と呼びます。環境変数を使うには **export** コマンドを実行する必要があります:

- export 変数名 [=値] — 環境変数を定義する

実は上に挙げたシェル変数のうち HOME、USER、PATH、TERM などは環境変数として宣言ずみですし、それ以外にもつぎのようなものが一般に使われます。

- $EDITOR — メールやニュースを打つのに使うエディタ
- $PAGER — メールなどを表示するのに使うページャ
- $PRINTER — 標準のプリンタ
- $MANPATH — man コマンドがマニュアルを検索してくれる場所

環境変数の一覧を表示させるのには **printenv** コマンドが使えます:

- printenv — 環境変数とその値の一覧を表示

2.7 ドットファイル option

ここまでに出てきたシェル変数や環境変数などの設定を、ログインするたびにいちいち打ち込むのでは大変すぎます。そこで、ホームディレクトリに**.bashrc** というファイルを書いておくことで、これらの設定を自動的に行わせることができます。[11]たとえば筆者のサイトでユーザに配布している標準の.bashrc はつぎのような内容です:

[11]実際には bash ではシステム全体の設定ファイルやログイン時/ログアウト時固有の設定も別のファイルで指定可能ですが、話が難しくなるので略しました。

```
ulimit -c 0     ←コアダンプファイルを作らないように
umask 077       ←標準のファイル保護モードは「go-rwx」
PS1="% "        ←プロンプトの設定
export PATH=/usr/local/bin:/usr/bin:/usr/sbin:/bin/sbin ←パス設定
export EDITOR=emacs-nw     ←標準のエディタ
export PAGER=/usr/local/bin/less  ←標準のページャ
export SHELL=/usr/local/bin/bash  ←標準のシェル
export ESHELL=/bin/sh     ← emacs のシェル窓用のシェル
export NOMHNPROC=1        ← mh の設定
export METAMAIL_PAGER=/usr/local/bin/less  ← metamail の〃
```

最初の 3 行を除いて大部分がシェル変数/環境変数の設定になっていますね。.bashrc の内容は自分の好みに応じて調整できるようになっておくのがよいでしょう。なお、.bashrc は変更しただけではその変更はすぐには効果を表さないので、設定内容を吟味するには **source** コマンドでそれを読み込ませてみる必要があります:

- source ファイル — ファイルをシェルに直接実行させる

「直接実行」とはどういう意味でしょうか? 実はファイルをシェルに実行させる標準的な方法は、新しいシェルのプロセスを起動し、そのシェルにファイルの内容を読み込ませて実行させることです。[12]しかし、別のプロセスを起動してそこの環境を設定しても、その設定はプロセスが終了した時になくなってしまいます。そこで、現在使っているシェルに「直接」ファイルの内容を実行してもらうのに source コマンドを使う必要があるわけです。

3 ユティリティとフィルタ

3.1 「大きなユティリティ」と「小さなユティリティ」

ユティリティというのは、現実世界ではおおむねキッチンなどの横にあって洗濯やアイロン掛けなどができるようなスペースを指すようですが、計算機の世界では「ファイルの形式変換など汎用的な操作をやってくれる便利なプログラム」といった程度の意味で用いられます。

そして、「小さな政府」「大きな政府」という概念があるのと同様に、ユティリティにも「大きなユティリティ」と「小さなユティリティ」があります。大きなユティリティとは

- 1 つのユティリティプログラムに、沢山の機能がついていて、何でもできてしまうことをめざすもの

のことです。たとえばアーミーナイフ (図 13 右) のようなもの、と考えればいいでしょう。それ 1 つで何でもできるというのは便利そうではありますが、その代わりつぎのような弱点があります:

- どの機能を使うかといった指定が沢山必要で、使い方が複雑になりがちである。
- どれか 1 つの機能だけ使いたいときでも全機能を備えたプログラムが動くので遅くなりがちだし計算機資源の無駄づかいである。
- 機能をちょっと増やすとか訂正するといったことは、その巨大なプログラムを直さなければならず面倒だし実際上不可能なこともある。

これに対して、小さなユティリティというのは

[12]その方が、ファイルの内容に誤りなどがあった時に現在使っているシェルに影響が及ばないので安全ですから。

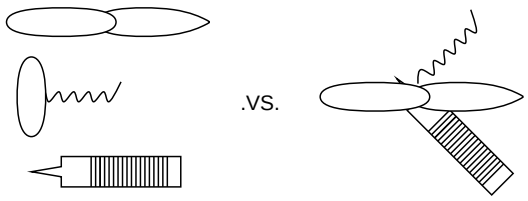

図 13: 単機能の道具と多機能の道具

- 1 つのユティリティプログラムは 1 つの (単純な) 機能しか備えていない

ものを意味します。それではあまり使えなさそうに思えるかも知れませんが、複雑なことをやりたい場合には小さなユティリティを複数組み合わせて使えるように設計するわけです。こちらの利点は前の裏返しです:

- 1 つのユティリティはごく単純で使い方もすぐわかる。
- 使いたい機能に対応するユティリティだけ動かせば済む。
- 足りない機能があったら、その機能だけを行う小さなプログラムを書いて追加すれば既存のユティリティと組み合わせて使える。

3.2 フィルタ

Unix は伝統的に「小さいユティリティ」の文化を持っています。「小さいユティリティ」のためには複数のプログラムを組み合わせて動かす仕組みが必要ですが、Unix にはそれが最初から備わっていたから、というのが大きな理由です。具体的には、先に上げた**パイプライン**記法を使います:

プログラム `1` | プログラム `2` | … | プログラム `N`

途中にあるプログラムはどれも「標準入力から入力データを読み取り、処理を行なって、標準出力にデータを書き出す」という形で動作します。その形がちょうど、空気や水をろ過するフィルタに類似しているので、この種のプログラムのことを Unix では**フィルタ**と読びます。

なお、このパイプラインに投入するデータは「プログラム 1」に入力リダイレクションで与えて、「プログラム N」から出力リダイレクションでどこかに保存すればいいので、これらのプログラムもフィルタであって構いません。しかし「プログラム 1」はデータを生成する一方のプログラム (たとえば `ps` など) であってもよいし、「プログラム N」はデータを消費するだけのプログラム (たとえば 1 画面ずつ表示するプログラム `less` など) であってもよいわけです (`less` については後の節で再度説明します)。なお、Unix の多くのフィルタは、コマンド引数にファイル名を与えると標準入力の代りにそのファイルからデータを読み出すようになっています (つまりフィルタとしてもデータ生成型プログラムとしても動作するわけです)。

ところで、このようなパイプライン処理がうまくいくためには、どのプログラムの出力も別のプログラムの入力として役立つようになっている必要があります。例えば Word 形式のファイルは一太郎では読めない、とかいったことをやっていては駄目なわけです。

では、計算機の世界で汎用的に使えるデータ形式とはどんなものでしょうか? 色々な考え方があり得ますが、Unix の場合それは「テキストファイル」つまり人間が読み書きできるようなファイルなら何でも、という方針を取っています。そうしておけば、データを人間が用意するのも楽だし (エディタで打ち込めばよい)、途中結果もそのまま画面に表示して調べることができるわけです。

次の節では代表的なフィルタをいくつか見ながら、これらの原理がどのように具体化されているかを学ぶことにしましょう。

3.3 cat option

cat は初心者向けには「ファイルを画面に表示する」コマンドだと説明することが多いのですが、実際には「入力を出力にコピーする」フィルタであり、ただしファイルを指定するとそのファイルがコピーされ、出力が画面につながっている場合には画面に表示されるため、ファイルを画面に表示するのにも使えるようになっています。実は cat で入力ファイルを複数指定した場合、それらをつなげて出力するので、ファイルの連結 (conCATination) に使う、というのが本来の用途です。

- cat ファイル… — ファイルや標準入力を連結して出力

また、ファイル名の代わりに「-」を指定すると、標準入力から読み込んだものがそこに入るので、パイプラインを流れるデータの前後に何かをつけ加えるのにも使えます:

```
% cat line
--------------------------------  ←ファイル line の中身は横線
% ps | cat line - line            ← ps 出力の前後に横線をつける
--------------------------------
PID TT STAT TIME COMMAND
13038 p0 IWs+ 0:00.54 -csh (tcsh)
12116 v0 IWs 0:00.75 -usr/local/bin/tcsh
12990 v0 IW+ 0:00.23 xinit
13005 v0 IW 0:01.41 twm
--------------------------------
%
```

そのほかに、データを「加工」するためのオプションも少々あります:

- -v — 制御文字などを見える形で打ち出す。ファイルの一部に制御文字が入っていて悪さをするらしいがよくわからない、といったときに便利。制御文字だらけのファイルなら od を使ったほうがよい。
- -n — 各行に行番号をつけ加える。どうしても行番号つきで打ち出せといわれたとき思いだしましょう。

3.4 tr option

tr は入力の各文字を (原則として)1 対 1 で別の文字に変換 (TRanslate) して出力するフィルタです。たとえばキーボードが壊れていてどうしても小文字の「a」が大文字になってしまうマシンでファイルを打ち込んだとしましょう (何てわざとらしい例!)。あとで大文字の「A」を小文字に直すのには次のようにします:

```
% cat test.txt
ThAt is A cAt.
% tr A a <test.txt
That is a cat.
%
```

tr の基本的な使い方は次のとおりです:

- tr 文字列$_1$ 文字列$_2$ — 文字を別の文字に変換

つまり「文字列$_1$」の 1 文字目は「文字列$_2$」の 1 文字目、2 文字目は 2 文字目、というふうに対応する文字どうしの変換が行われるわけです。ほかの多くのフィルタと違って、tr だけはファイル名指定を受け付けないので、ファイル入力が必要なら入力リダイレクションを用います。

しかし、注意深い人は「そのキーボードは小文字の「a」が入らないんじゃなかったのか」とおっしゃるかも知れませんね。別のマシンに移ったんだよ、という言い訳もあり得ますが、どうしても壊れたマシンでやりたければ、任意の文字を 8 進文字コードで指定することができます (文字と 8 進コードの対応は「man ascii」で見てください)。

```
% tr A '\141' <test.txt
That is a cat.
%
```

もうちょっと実用的なのは、すべての大文字を対応する小文字に直すことでしょう。

```
% tr A-Z a-z <test.txt
that is a cat.
%
```

もちろん、これは

```
tr ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
```

と同じ意味で、連続する文字範囲を簡単に指定するために「-」が使えるようになっているわけです。ここまでは指定する 2 つの文字列の長さが同じでしたが、もし文字列 2 のほうが短ければ、文字列 1 のあふれた文字には文字列 2 の最後の文字が対応します:

```
% tr A-Za-z a <test.txt
aaaa aa a aaa.
%
```

tr にもいくつかのオプションがあります。[13]

- -c — 文字列 1 に「ない」すべての文字を集めたものを改めて文字列 1 だと思う。

例を見てみましょう:

```
% tr -c A-Za-z /<test.txt  ←英字以外の文字をすべて「/」に
ThAt/is/A/cAt//% ←改行文字が「/」になったので改行されない
```

「すべての」文字に改行文字も含まれるため、改行文字まで「/」になってしまいました。これがいやなら次のようにします:

```
% tr -c 'A-Za-z\012' /<test.txt  ← 012 は改行文字の 8 進コード
ThAt/is/A/cAt/
%
```

- -d — 文字列 1 に現われる各文字を消去する (delete)。この場合、文字列 2 は指定する必要がない。

こちらは不要な文字を削除するのに便利です:

[13] このあたりは Unix のバージョンによって違いがあります。

```
% tr -d ' ' <test.txt  ←空白文字をすべて削除
ThAtisAcAt.
%
```

Windows からファイルをもってきた場合、行末に 015(キャリッジリターン文字) が余分についているため、「tr -d '\015'」を使って取り除く、というのが常套手段です。

- -s — 置き換えが連続して起こる場合には、最初の 1 個だけ出力 (squeeze)。

これも使い方が分かると便利です。たとえば単語の「数だけ」知りたい場合は、次のようにすればいいわけです:

```
% tr -s A-Za-z a <test.txt
a a a a.
%
```

これらのオプションを組み合わせると、いろいろ役に立つフィルタになります。たとえば「tr -cd 'A-Za-z \012'」は、英字と空白と改行以外のすべての文字を消してしまうので「きれいな」テキストファイルが作れますし、「tr -cs A-Za-z '\012'」は英字以外のものの並びをすべて改行文字 1 個に置き換えるので、各単語を 1 行ずつバラバラにしたファイルが作れます。このように tr は機能自体は単純なのにアイデア次第でいくらでも応用が効く、「フィルタの鑑」だといえます。

3.5 grep 族

grep、**fgrep**、**egrep** の 3 つのコマンドはいずれも「入力のなかにあるパターンを含む行があったら、その行全体を打ち出す」という機能を提供する同類のフィルタです (指定方法は 3 つとも同じ):

- grep パターン ファイル… — ファイル中でパターンを含む行を出力

オプションも次のものが 3 つ共通に使えます。

- -v — パターンを「含む行」の代わりに「含まない行」を打ち出す。
- -n — 行を打ち出す際に行番号を一緒に打ち出す。

そして、3 つの違いはパターンとして何が書けるかの違いになります。まず fgrep の場合、パターンとしてたんなる文字列のみが書けます。たとえば「that」をいつも「taht」打ってしまうくせがある人が、そのまちがいをチェックしたければ次のようにします:

```
% fgrep taht wrong.txt
What is taht?
%
```

しかし、「That」のように文の頭にくるときは大文字なのでこれではみつかりません。そのような場合には grep のパターンを使えばよいのです:

```
% grep '[Tt]aht' wrong.txt
Taht is a cat.
What is taht?
%
```

「[…]」はシェルのファイル名置換と同様、「…」の部分のどれか 1 文字にマッチするパターンです。なお、「[」と「]」はシェルのメタキャラクタなので、grep にパターンとして渡すときには「'…'」で囲む必要があります。では、fgrep の存在意義は何でしょうか? それは、たとえば「[」という字を探したければ fgrep で探すほうが簡単なわけです。grep で探したい場合には「\[」のように前に「\」をつけなければなりません。

さて、that だけでなく this も thsi と打ってしまう人が両方探したい場合はどうでしょうか。そのときは grep でも力不足で、egrep で次のように指定します:

```
% egrep '[Tt](aht|hsi)' wrong.txt
Taht is a cat.
What is taht?
Thsi isn't a dog.
% ...
```

丸かっこは「くくり出し」を、縦棒は「または」を表わしています。この丸かっこと縦棒が egrep で加わった機能なわけです。ここでパターンについてまとめておきましょう。

- c — c という文字そのもの。
- [⋯] — ⋯のうちどれか 1 文字
- . — 任意の 1 文字。
- ^α — 行の先頭の α
- α\$ —行の末尾の α
- α? — α または空。(1)
- α* — α というパターンの 0 個以上の繰り返し。(1)
- α+ — α というパターンの 1 個以上の繰り返し。(1)
- (⋯) — くくり出し。(2)
- α | β — α または β。(2)
- \(⋯\) — ⋯のところを一時的に覚える。(3)
- \1、\2 — 覚えたものの 1 番目、2 番目、⋯。(3)

(1) は grep ではパターン α が 1 文字に対応するパターンでなければならないという制約があります。(2) は egrep のみの機能です。一方 (3) は grep のみの機能です。また、「.」の長い連続など、組み合わせが爆発的に多くなるパターンは egrep では実現上うまく扱えません。というわけで、grep と egrep は適材適所で使い分ける必要があるわけです。

おもしろい練習として、/usr/share/dict/words という英単語がたくさん入ったファイルからパターンに合った単語を取り出してみるとよいでしょう (less を使うのは、たくさんあてはまったとき、1 画面ずつ止まりながら見るためです):

```
% grep 'パターン' /usr/share/dict/words | less
```

おもしろそうなパターンの例をあげておきます:

'[aeiou][aeiou][aeiou]'	母音が 3 つ続く単語
'tion$'	末尾が tion で終わる単語
'^z'	z で始まる単語
'^.........$'	長さ 10 文字の単語
'\(...\)\1'	3 文字の反復を含む単語
'\(.\)\(.\).\2\1'	5 文字の回文を含む単語

もちろん、うろ覚えの単語を探すという実際的な使い方もあるわけです。

3.6 sed option

tr による文字置換はとても強力ですが、しかし「自分はどうしても that を taht と打ってしまうので、これを正しく直したい」といった仕事には無力です。というのは、tr は入力の各文字をバラバラに扱うので、特定の 2 文字の組みを置き換えるのには使えないからです。連続した文字列の置き換えには **sed**(stream editor) が適役です:

- sed コマンド ファイル… — コマンドに従って入力を加工する

たとえば上の例題は次のようにしてできます:

```
% cat wrong.txt
What is taht?
% sed 's/taht/that/' wrong.txt
What is that?
%
```

この「s」(substitute) コマンドだけ覚えておけばほとんど十分でしょう (ほかのコマンドについては man ページで調べてみてください)。なお、ただの s コマンドは 1 行に 1 回しか置き換えを行ないませんが、taht が全部 that になるまで繰り返しやりたければ「's/taht/that/g'」のように末尾に「g」をつけてください。

たったこれだけ…? と思うかもしれませんが、実はこの「s」コマンドによる置き換え指定のなかには grep と同じパターンが書けるので、これだけでかなり強力な修正ができます:

```
% cat test.txt
a 21
is 10
this 3
% sed 's/\(.*\) (.*\)/\2 \1/' test.txt
21 a
10 is
3 this
```

これはどう読むかというと、「入力行を任意の文字列 1 と、空白と、また別の任意の文字列 2 にマッチさせ、それ全体を 2、空白、1 の順でつなげたものに置き換える」という意味になるのです。

場合によっては、こういう置き換えを多数やりたいかもしれません。その場合は

```
sed -e 's/Thsi/This/g' -e 's/Taht/That/g'
```

のように各コマンドのまえに-e オプションをつければ、いくつでもコマンドが書けます。あるいは、

```
s/Thsi/This/g
s/Taht/That/g
```

のように多数のコマンドを並べたファイルを準備しておき、「sed -f ファイル」の形で指定することもできます。sed は入力の各行についてファイルのなかにある命令を 1 行ずつ「実行」してくれます。つまり、これは一種の「プログラム」なわけです。

3.7 sort と uniq option

sort はファイルの行を指定した順番に並べ替えるコマンドです。

- `sort` ファイル… — 行単位での並べ替え

いちばん簡単には、何もオプションを指定しないと、`sort` は行全体を比較して、その文字コード大小順にもとづき、行を小さい順に並べます:

```
% cat test.txt
this
is
a
pen
% sort test.txt
a
is
pen
this
%
```

しかし、文字コード順だと $A < B < \ldots < Z < a < b < \ldots < z$ なので、大文字と小文字が混ざっているとあまり嬉しくないかも知れません:

```
% cat test.txt
This
is
a
pen
% sort test.txt
This
a
is
pen
%
```

このようなときには「`-l`」(letter:英字) オプションを指定すれば $a < A < b < B < \ldots < Z < z$ の順番にしてくれます。また、数値データも文字コード順で比較されるとあまり嬉しくありません:

```
% cat test.txt
10
2
1
% sort test.txt
1
10
2
%
```

つまり、文字コード比較だと「1」で始まるものが全部終わってからはじめて「2」で始まるものが来ます。数値の場合には「-n」(number) オプションを指定することで、数値としての順に並べてくれます。さらに、いつでも「-r」(reverse:逆) オプションを追加することで小さい順ではなく大きい順に並べられます。

行全体ではなく、行の特定の部分にもとづいて並べ変えを行わせることもできます。その指定方法は少し面倒ですが、いちばん簡単には「+0」「+1」などと指定することで「最初の欄」「2 番目の欄」などを指定できると覚えておけばよいでしょう。より正確には「man sort」でマニュアルを調べてみてください。

ふたたび、行全体を整列する場合に戻りますが、「どのような行があるか」だけを知りたい場合には重複を除く (つまり同じ行が複数あった場合に 1 行だけ残してあとは消してしまう) ほうが望ましいですね。一般のファイルについてこれをやるのはたいへんそうですが、整列したあとなら同じ内容の行が隣り合っているので簡単にできます。それをやってくれるのが **uniq** というフィルタです:

- uniq ファイル… — 整列後のファイルの重複を除く

たとえば単語リストでこれを行ってみましょう:

```
% cat test.txt
this is a pen.
what is this?
% tr -cs A-Za-z '\012' <test.txt
this
is
a
pen
what
is
this
% tr -cs A-Za-z '\012' <test.txt | sort
a
is
is
pen
this
this
what
% tr -cs A-Za-z '\012' <test.txt | sort | uniq
a
is
pen
this
what
%
```

なお、uniq に-c(count) オプションを指定すると、同じ行がいくつずつあったか数えてくれます:

```
% tr -cs A-Za-z '\012' <test.txt | sort | uniq -c
 1 a
 2 is
```

```
   1 pen
   2 this
   1 what
%
```

3.8 そのほかのよく使うフィルタ類

あと少しだけ、簡単なフィルタとフィルタの末尾に使うプログラムについて簡単に説明しておきます:

- **head** -行数 — 先頭から指定した行数のみ取り出す
- **tail** -行数 — 末尾から指定した行数のみ取り出す

これらはファイルの先頭 N 行または末尾 N 行だけをもってくるフィルタであり、長いファイルの頭のほうだけ、ないし終わりのほうだけみたい場合に使います。行数を省略すると 10 行分取り出されます。また、入力の行数が指定行数より少ない場合はその全部が取り出されます。

フィルタでファイルを加工していて、その結果が長い時に「流れて行かずに」見たい時には **less** を使うと便利です。

```
フィルタによる処理… | less
```

`less` は、表示する行が画面一杯になるか終わりになると、そこで表示を一時停止します。次の 3 つのコマンドを使って表示を制御してください。

```
[SP] --- 1 画面ぶん先へ進む
b    --- 1 画面ぶん前へ戻る
q    --- 表示を終了する
```

最後は必ず「`q`」を使って終わる必要があることに注意。また、他のフィルタと同様にファイルを指定して「長いファイルを 1 画面ずつ眺める」のに使うこともできます。

最後に、結果を表示するのでなく、単に文字数や行数などが知りたいだけのときは、**wc** (Word Count) を使ってください。

```
% ps a | wc
     7      39        241
行数↑       ↑単語数   ↑文字数
```

3 つの数字が表示されますが、これは順に「行数」「単語数」「文字数」です。「`wc -l`」「`wc -w`」「`wc -c`」のようにオプションを指定することで、それぞれ行数、単語数、文字数だけを出力させることもできます。

4 シェルスクリプト

4.1 スクリプトとは

現在の計算機システムで重要になっている考え方の 1 つに「**スクリプト**」があります。スクリプトとは、もともとは「簡単に書けるプログラム」くらいの意味でしたが、現在ではかなり高度なシステムでも、それに適したスクリプト言語を選ぶことで簡単に実現できます。

スクリプトの英語本来の意味は「台本」であり、計算機の世界では「実行しなければならないことを順番に書き連ねたもの」という意味で使われてきました。しかし、動作を順に書いたもの、とはプ

ログラムなわけであり、つまりプログラムを書くことで 1 つひとつ人間が指示しなくてもいいようにしましょう、というとても当り前な (しかしとても重要な) ことを意味しているわけです。

たとえばGUIを使ったツールである操作をやるのに、マウスで範囲を選択し、再度マウスでメニューを出して操作を指定し、それで完了、簡単でいいね、と思っていたとします。しかしその操作を 100 個のファイルについてやらなければならないとしたら、またはテキストのあちこちにある 100 箇所についてやらなければならないとしたら、それは苦行になってしまいます。100 ならまだましで、1,000 だったら、10,000 だったらどうでしょう? あなたはまだその苦行を続ける気がありますか?

もちろん、そういう苦行から人間を開放することこそ計算機の本来の存在意義なわけで、計算機のためにそんな苦行をするというのは本末転倒以外の何者でもありません。ではどうすればいいのでしょうか? 同じことを繰り返すのは計算機の得意技なのですから、プログラムを書いてプログラムにやらせればよいのです。

しかし自分はプログラマではないからプログラムなんか書けない、プログラマに頼むとひどく時間が掛かって高いわりに全然こちらの要求と違うものができるだけ、ですか? 後半のソフトウェア工学的問題はさて置いて、[14]あなたに「書けない」のは単にあなたが「正しい道具」を使っていないから、かも知れません。

プログラミング初心者がCやC++など「プロがソフトウェア開発するための道具」として作られた言語を瞬時に使いこなせるようになる、というのは難しいでしょう。[15]しかし世の中は進歩していて、そういう「プロっぽい道具」以外に、もっと楽に、ちょっと書くだけで望むことができるようなプログラムの世界、というのがかなり発達してきています。それがスクリプト (とスクリプト言語) の世界、というわけです。本当にそんな夢のような世界があるのかどうか、どれくらいが夢でありどれくらいが現実なのかについて、これから見て行きましょう。

4.2 対話的シェルからシェルスクリプトへ

ここではスクリプトの原点であるシェルスクリプトを少しだけ見てみます。Unix ではプログラムの標準入力がふだんはキーボードに接続されていますが、入力リダイレクションによってこれをファイルに切り替えたりできることはすでに学びました。ところで、シェルはふだんキーボードからコマンドを読み込んでいますが、これをファイルに切り替えたらどんなことが起こるでしょう?

```
% cat pswho     ← pswho というファイルの中身は次の通り
ps
who
% bash <pswho  ← bash にそのファイルを入力として与える
  PID TT STAT  TIME COMMAND
29493 p0 IW    0:00 /usr/local/bin/bash
  330 p1 S     0:00 bash
  331 p1 R     0:00 ps
29510 p1 S     0:06 /usr/local/bin/bash
kuno     ttyp0   May 23 11:03   (smri02:0.0)
kuno     ttyp1   May 23 11:04   (smr03:0.0)
%              ↑「ps」と「who」が実行された
```

つまりファイルからコマンドが読み込まれて実行されるわけです (図 14)。

[14]「ソフトウェア工学」とは、ユーザが必要とするソフトウェア (ないしシステム) を、必要な期間内に、必要なコスト以下できちんと実現する方法を探求する学問分野です (単なる「プログラムの作り方」はソフトウェア工学には含まれません)。

[15]本書ではここまでに C 言語の例題がいくつも出ていますが、これは「計算機のしくみを学ぶ手段」として扱っているのであり、自分が望む道具を作るとなると、質の違った努力が必要になります。

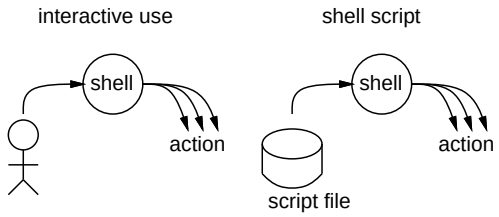

図 14: シェルスクリプトの概念

このように、キーボードから打ち込む代わりに、あらかじめファイルにその内容を用意しておき、それを自動的に順次実行させるようなものがシェルスクリプトであり、スクリプトの「原点」となっています。実は、シェルもほかのフィルタと同様、入力ファイルが指定されていたら標準入力の代わりにそのファイルから入力するので、リダイレクションを使う代わりに単に

```
% bash pswho
```

としても同じです。

さて、シェルスクリプトはいったい何の役に立つのでしょう? いちいちファイルに打ち込むよりはいきなりキーボードからコマンドを入れる方が簡単だと思いますか? スクリプトにはたとえば次のような利点があります。

- 複雑なコマンド列を用意する時に、エディタを使って少しずつ試しながら作っていくことができる。
- 何回も繰り返し使う長いコマンド列をそのつど打ち込まなくても済む。
- 計算機にコマンド列を生成させてそれを実行させられる。

たとえば、ログイン時の設定を行う.bashrc もシェルによって自動実行されるので、シェルスクリプトの仲間です。つまり、毎回決まった初期設定などを行うためにスクリプトを使用しているわけなのです。

4.3 指令としてのシェルスクリプト

さらに面白いのは、シェルスクリプトを格納したファイルを chmod によって「実行可」にしてやるとそのファイルは「シェルによって実行されるプログラム」になる、ということです:

```
% chmod +x pswho     ←ファイル pswho を「実行可能」に
% ls -l pswho
-rwxr-xr-x  1 kuno            7 May 23 13:43 pswho
% pswho              ←ファイル名を打ち込むと
  PID TT STAT  TIME COMMAND
29493 p0 IW    0:00 /usr/local/bin/bash
  335 p1 S     0:00 /usr/local/bin/bash
  336 p1 R     0:00 ps
29510 p1 S     0:06 /usr/local/bin/bash
kuno     ttyp0   May 23 11:03   (smri02:0.0)
kuno     ttyp1   May 23 11:04   (smr03:0.0)
%                    ↑さっきと同様に実行された
```

もちろん、このファイルが$PATH に指定されるコマンドディレクトリのどれかに置かれていれば、そのファイル名を打ち込むだけでどこからでも実行できます。このようにして、いちいち C 言語などでプログラムを書かなくても、自分用の新しいコマンドを増やせるわけです。

ところで、ここで疑問なのは、Unix にはシェルは多数あったということです。このようにして用意したシェルスクリプトは一体どのシェルによって実行されるのでしょうか? 特に指定がなければ /bin/sh (Bourne シェル) が使用されますが、どれか特定のシェルを使用したい場合にはファイルの 1 行目に

```
#!シェル名 引数 …
```

という形のものを入れておくことで、実行用シェルを指定できます。これを**インタプリタ指定**と呼びます。一般に、実行可能なファイルの先頭に「#!コマンド 引数 …」と書かれていた場合、そのファイルをコマンドとして実行すると

```
% コマンド 引数 … ファイル名 [RET]
```

と同じ動作が行われます。「コマンド」は特にシェルのプログラムでなくても構いません。

4.4 スクリプトの引数と変数

ところで、シェルスクリプトによってコマンドが作れるとしても、それに対してさまざまなオプションや引数を渡せなければあまり面白くはありません。実は、スクリプトをコマンドとして起動したときに、その 1 番目、2 番目、... の引数の値は$1、$2、... というシェル変数に予め設定されています。従って、それらを参照したスクリプトを書くことにより、引数の値を活用できます。たとえば、つぎのシェルスクリプトは指定したファイルについての ls -lg を実行します (引数が指定されなかった場合には、$1、$2、... はすべて空なので通常の ls -lg と同じになります)。

```
% cat lg
#!/bin/sh
ls -lg $1 $2 $3 $4 $5 $6 $7 $8 $9
% lg abc t.c
-rwxr-xr-x  1 kuno     faculty        14 May 23 13:47 abc
-rw-r--r--  1 kuno     faculty        31 May 14 14:16 t.c
%
```

このままだと引数の数は最大で 9 個までしか参照できませんが、それでは不便なので、かわりに$*と書くことですべての引数をそこに埋め込むことができます。

さらに、ここでは説明しませんでしたが、シェルスクリプトの中で繰り返しや枝分かれや計算を行うこともでき、これらを組み合わせることで複雑な処理を行うプログラムを書くことも十分可能になっています。

5 まとめと演習問題

この章では計算機のソフトウェアの構造について、オペレーティングシステム (OS) の役割りをプロセス管理を中心に学びました。続いてユーザの意思を OS に伝える方式について検討し、Unix のシェルについてその機能を概観しました。さらに、シェルの機能を使って複数のユティリティを組み合わせる考え方や、シェルスクリプトの考え方についても学びました。

2-1. 「xclock -analog -update 1 &」により秒針付きの時計の窓を作り、psを使ってこの時計のプロセス番号を調べなさい。また、このプロセスを凍結したり再開したり強制終了するとどうなるか調べなさい。emacsやブラウザやその他の種類の窓だとどうですか? これらのプロセス凍結中にそのプログラムを使おうとするとどうなりますか? 再開するとどうなりますか?

2-2. さまざまなパラメタでpsコマンドを実行し、どのようなプロセスがあるかを観察しなさい。また自分のUIDが何番かも調べなさい。とくにPIDとPPIDをチェックして、プロセスの親子孫関係のグラフを描いてそこから何が分かるか考なさい (プロセスの作り方— コマンドを打ち込んで作るか、メニュー等で起動するか等 — によって親子関係のできかたが変わるはず)。

2-3. hello.cを打ち込んで動かしなさい。その名前を「a.out」から別のものに変更して、その名前で動くことも確認しなさい。もし既にあるコマンド (「ls」など) と同じ名前にすると。「ls」と言った場合どうなるか予想し、続いて試しなさい。なせそのようになるのか考えること。

2-4. Emacsを起動するのに、メニューを使う方法とコマンドで起動する方法のそれぞれがどれくらい時間を要するか、まず予測し、続いて測ってみなさい。具体的には:

(1) 「date[RET]」で時刻を表示させる。
(2) 「emacs &」または「背景右ボタンメニュー→ Emacs」でEmacsを起動させる。
(3) 再度「date[RET]」で時刻を表示させる。

この時刻の差 (所要時間) と、(1) と (3) だけを行った場合の時刻の差 (所要時間) とのさらに差を取ると、(2) だけに要する時間が分かります。数回ずつやって平均を取ること。

2-5. 英単語が多数入ったファイル (たとえば/usr/share/dict/words) を材料に、つぎの項目から3つ以上調べてみなさい (各種フィルタを組み合わせてやる):

a. 途中に「otion」が含まれている単語の数 (と代表例)。[16]
b. 「aho」というつづりと「ya」というつづりが両方含まれている単語。[17]
c. 末尾が「otion」で終わる単語で、「e」が含まれないようなもの。[18]
d. 先頭が「z」で最後が「tion」で終わる単語。[19]
e. 一番最後に出て来る単語。[20]
f. ちょうど1000番目に出て来る単語。[21]
g. 母音を5つ連続して含む単語。[22]
h. 5文字のまったく同じ文字の並びが2回出て来る単語。[23]

2-6. うろ覚えの英単語を探すために/usr/share/dict/wordsの指定した (grepの) パターンにマッチする語を1画面ずつ止まりながら表示するシェルスクリプトを「wd」という名前で作りなさい。使い方は次のような感じになる:

```
% wd abs
absampere
```

[16]ヒント: 「otion」の後ろに任意の文字が1つあればいいわけですね。数えるのはもちろんwcを使います。
[17]ヒント: まず「aho」が含まれているものを取り出し、その出力の中からさらに「ya」が含まれているものを探します。
[18]ヒント: 「終わる」は、grepの$の機能を使います。その後でパイプで「-v」つきのgrepで「e」を排除すればいいのです。
[19]ヒント: 途中に任意文字が0個以上あるわけです。
[20]ヒント: もちろんtailを使います。
[21]ヒント: こちらはheadとtailを組み合わせれば…
[22]ヒント: 母音とは「a」「e」「i」「o」「u」のどれかですね。
[23]ヒント: 5文字の並びを覚えて、そのあとに任意文字が0個以上あり、その後で覚えた並びがあればいいですね。

```
absarokite
abscess
abscessed
(止まるので SP を打つ)
abscession
abscessroot
abscind
abscise
(やめたければ q を打つ)
%
```

(ヒント: 「grep "$1" /usr/dict/words | less」という中身のシェルスクリプトを作ればよいはず。)

さらにオプションまで自習した方むけの課題ですが、このシェルスクリプトを自分用のコマンドとしてどこからでも使えるように登録してみるとなおよいでしょう。(ヒント: 自分用のコマンドディレクトリがなければ用意し、コマンドパスにも入れる。そして作成したシェルスクリプトを実行可能にしてこのコマンドディレクトリに置く。)