プログラミング環境 第1回

久野 靖 *

1991.10.1

1 はじめに

1.1 プログラミング環境とは?

まず、このレクチャーの題名である「プログラミング環境 (programming environment)」とは、何のことだと思うか?

```
programming  --- 分かります、よね?
environment  --- 環境 → 環境問題、環境汚染、etc, etc...??
```

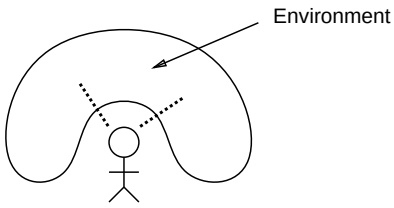

図 1: 「環境」の概念

つまり「環境」というのが問題そうである。取り敢えずは、図 1 にあるように、環境というのは...

自分のまわりで
自分と接していて
自分が働きかけたり
自分に働きかけてきたり ... するようなもの。

と、考えればいいのではないだろうか。

では、計算機に関わる「環境」というのは?具体的には、皆様が WS の前に座っている場合、もちろん直接接するのは画面、キーボード、マウスなどの入出力機器 (あ、もちろん椅子や部屋の空気もそうですが... まあそっちはいいですよね) だが、ご存知のとおり「計算機、ソフトなければただの箱」であるから、後ろでどんなソフトが (より正確にはソフトの「体系」だと思うが) 動いているかが肝心である。ソフトは椅子のように目に見えないし、空気のように肌で感じられないから「自分がどんな環境にいるか」を把握しづらいのだけど、実は計算機のソフトというのは自然現象よりはずっと単純だから、一旦その「体系」を把握してしまえばわりと簡単に使いこなして効率良く仕事が進められるようになるものである。

ところで、計算機環境というのも色々あるのだが、ここで主としてやるのは

*筑波大学経営システム科学専攻

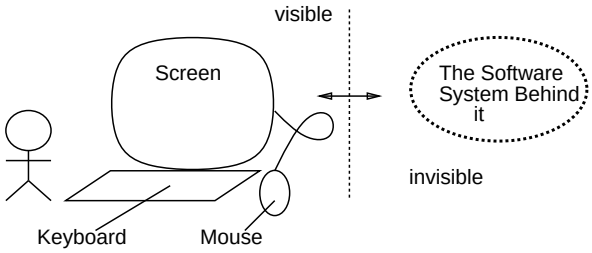

図 2: 目に見える機器と背後の体系

・プログラミング xxxxxxx 環境

であるといえる (xxxxxx の所には何が入ると思うか?)。典型的な答えは「を開発する」かと思うが。皆様が使っていて、ここでもとりあげる Unix + WS(または X 端末) などは典型的にそういうことをするのに適した環境だ、とされている。開発環境というのは実は

・プログラムを設計する
・プログラムをコーディングする
・プログラムをコンパイルする
・プログラムをテスト実行する
・もろもろの文書を書いたり管理したりする
・作成にあたってチーム内の連絡を取ったりする

などもろもろのことを効率良くこなすことを目指しているので、実は何にでもつかえるかなりゼータクな環境である (じゃそうじゃないのはどんなものがあると思うか?) まあ、そういう色々できる環境で勉強する、というのは正しい道であるわけだ。

1.2 プログラミング環境の歴史

ではいきなり現在のプログラミング環境の話に入る前に、どういう歴史をたどって現在に至ったかを簡単に復習してみよう。

1.2.1 有史以前

計算機がまだ売られてなくて、「作った人が使う」に近かった時代のこと。この辺はぜひ木村先生にコンパとかの席で聞いてみて欲しい。ちなみに木村先生は日本独自のパラメトロン計算機 PC-1 ができた頃まさにそれを作ったところにいらした人である。

1.2.2 オープンショップ制

要するに、時間割り当て制であって「私は何時から何時までこの計算機を使いたい」などとノートに書いたりして予約する。マシンにはボタン、スイッチ、ランプが沢山並んだ「操作卓」なるものがついていて、これでメモリやレジスタの内容を書き換えたり、実行を開始させたり虫とり用に 1 命令ずつステップさせたりできた。当然、これでは一人 1 台しか使えない。もちろんプログラムはマシン語かせいぜいアセンブラが当たり前であった。

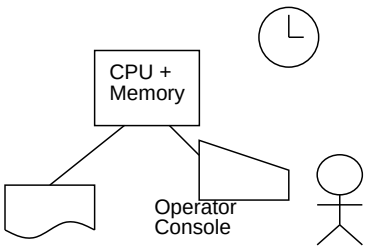

図 3: オープンショップの方式

1.2.3 バッチ時代

計算機は高価なのでそれを一時に一人ずつしか使えないのは困る、というので次にきたのがバッチシステムの時代である。今度はプログラマはプログラム (このころから Fortran などが使えるようになる) を巨大なタイプライタつき机みたいな「パンチ機」に座ってカードに打ち、それと翻訳や実行などの指示を書いた制御カードとをたばねて (ジョブ、とい呼ぶ) カウンタに提出する。オペレータと呼ばれる人が全員のカードをまとめて持っていってカードリーダから読ませると、順番に翻訳や実行が起こり、結果がラインプリンタに打ち出される。オペレータはそれを一人分ずつ切り離してカウンタの棚にのせて返却する。何を隠そう、私が情報科学科に入ったときはプログラミング入門はこれであった!

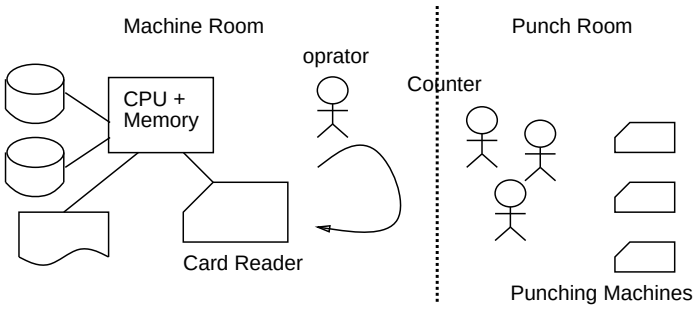

図 4: バッチ方式

1.2.4 TSS の時代

バッチ方式では一日せいぜい数回しかジョブが流せないので、計算機の効率は良くても人間の効率はえらく悪い。そこで計算機にオンライン端末をつないで、コマンドを入れるとただちにその実行が始まる、という画期的な方式が TSS である。これに移行した時はまるで天国のように感じたものだが、たくさんの人間が CPU を取り合う点に変わりはないので込んでくるとやたら反応が遅くなるという弱点がある。あと、画面と CPU の間は細いはりがねであるから大体字ばっかりの表示を見るはめになるというのも今にして思えば弱点である。

1.2.5 ワークステーションの時代

計算機の値段がますます安くなると、結局は一人に 1 台計算機を与えて自由に使わせるのがいい、ということになる。だから図 6 で前と違うのはどこかに一つ CPU があるのではなく、全ての画面の裏に CPU が隠れているということである。これが WS なわけであるが、単体のパソコンと

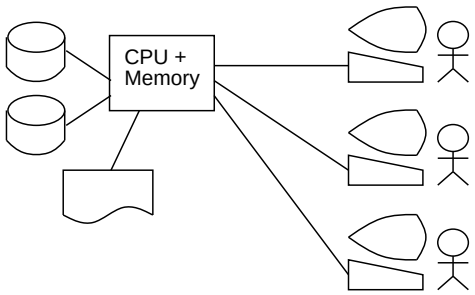

図 5: TSS 方式

も違ってファイルなど共有される資源はサーバと呼ばれる別の計算機に入っていて、それをネットワーク経由で利用する。従ってどの WS に座っても自由に自分のファイルが使えたりするわけである。また、WS を用いたシステムではの CPU 能力を活用して広い画面を提供し、字ばっかりでなく様々な図形や絵も駆使したユーザインタフェースが使えるようになっているのが普通である。

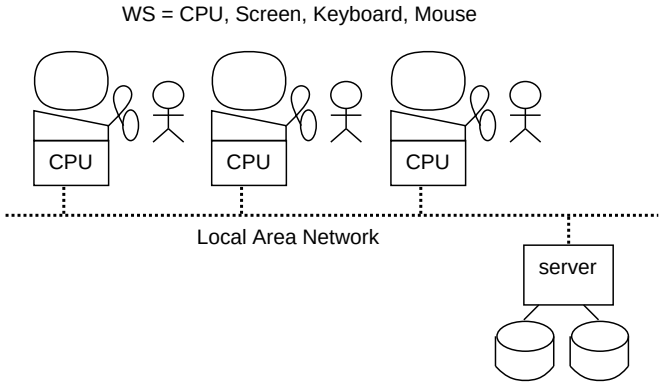

図 6: WS による分散システム

1.2.6 X 端末 — 廉価版の WS 環境

さて、昨年までだと話はこれで済んだのだが、今年から情報科学科の 2、3 年生の皆様が使うシステムはさらにもうちょっと進化 (?) した。つまり、WS というのはそれぞれがかなり強力な CPU を持っているので高価である。そこで、WS によるシステムがファイルサーバを共有していたように、CPU もサーバによる共有に改め、手元の WS は画面入出力だけをやるようにしたシステムが考えられるようになった。「画面入出力だけやる機器」というのは要するに「端末」であるが、ただし昔の端末と違うのは WS の「廉価版」であるから、この「端末」はちゃんと図形や絵を扱う機能を持っているし、サーバとのやりとりも高速なネットワークを使っているから快適に行なえる。なお、我々のところにある「端末」は、もともと WS の上で動かしていた「X-Window」と呼ばれるウィンドウシステムの画面入出力部分が動くように作られているので「X 端末」と呼ばれる。

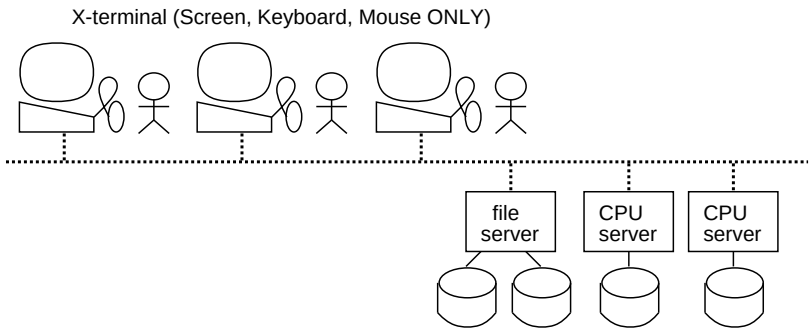

図 7: X 端末による分散システム

2 計算機システムと OS

2.1 計算機システムとは?

計算機システム、というのはそもそも何だろう。もちろん、「計算機」という言葉と「システム」という言葉から成っているわけだが。

計算機 (computer):
- 計算する? --- △
- 情報を、処理する。情報とは?処理するとは?
- bit 列を、加工する。

システム (system):
- 体系?
-「やりたい事」ができるように、構成要素を組み合わせたもの

ではもう少し具体的に見てみよう。

2.2 裸の計算機システムの構造

裸、というのは何もソフトウェアが載っていない、という意味。つまり計算機のハードウェアのみ。例えば皆さんが使うシステムの外見は図 1 のような感じである。もちろん、見た目だけで分かるわけにはいかない。そもそも一番めだたない「箱」が一番高価なのである。一方、ハード屋さんの目から見た計算機システムというのは図 2 のような感じに描かれることが多い。 ここで CPU(central processing unit) というのが一番偉く「情報を処理する」ところ。しかし、CPU の中に蓄えられる情報はあまり多くないので、主記憶というところに主に情報を蓄る (もちろん、この情報も bit 列)。CPU と主記憶だけだと処理する情報を外界とやりとりできないので、そのために入出力装置 (これが主として外に見える) が必要。入出力装置と CPU の間で直接情報をやりとりすることはまずなくて、CPU はコントローラというのにどこととこの間で転送せよ、などと命令することで入出力装置を制御する。

ハード屋さんにとってはこの通りに線がひっぱってあるから図 2 が一番自然だが、ソフト屋さんにとってはどちらかというと図 3 のように見える。つまり、CPU と主記憶は (自分の書くプログラムがそれらを駆使して走るから) ほとんど一体に感じらる。そして、利用者とやりとりするのが画面やキーボードやマウス、一方ディスクやテープやネットワークはプログラムの主記憶だけではたりない記憶を補う、というイメージである。

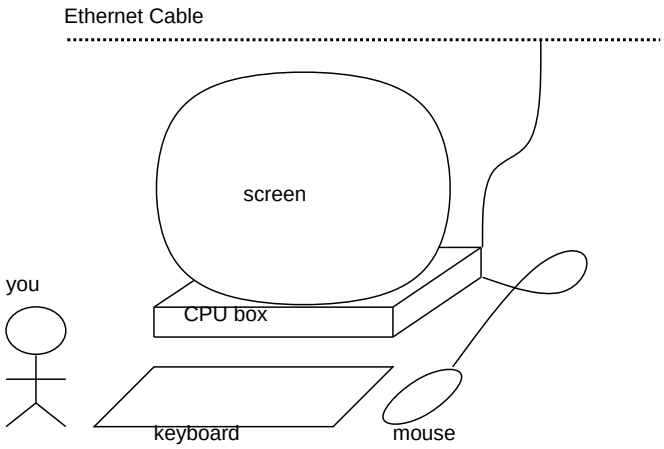

図 1: 外から見たとある計算機システム

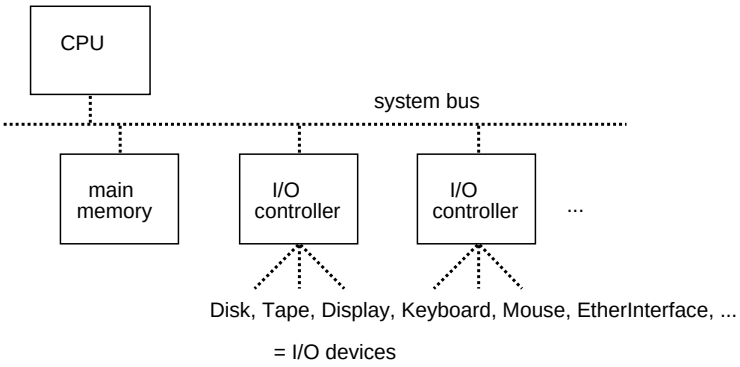

図 2: ハード屋さんから見た計算機システム

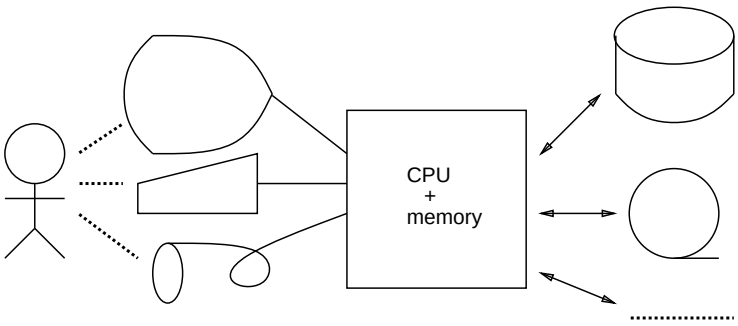

図 3: ソフト屋さんから見た計算機システム

2.3 ソフトウェアから見た計算機システム

では、今度はソフトの動いている計算機はどう見えるかを図 4 に示す。すごく簡単に見えるが... 計算機はハードだけではただの箱で、ソフトが動いて始めて人間とやりとりが出来る、ということを如実に示している。ここで、利用者のアクションと計算機からの応答のやり方の形態はいろ

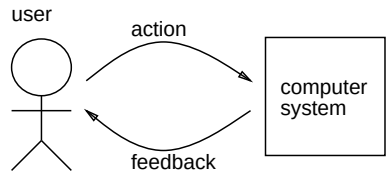

図 4: 動いている計算機システム

いろあるが、とりあえず一番ありふれた次のようなものを考える。

・`action =` キーボードから指令を打ち込む。
・`feedback =` 画面に何らかの表示が起きる。

ところで、なぜそんなことが可能になるのだろう?(間)
それは... 先に、CPU は「情報を処理する」=「bit 列を加工する」能力を持った所だ書いたが、実は「どのように bit 列を加工するかを記した命令を解釈実行する」能力を持つ、というのが正しい。では、その命令はどこにあるかというと... それは主記憶に格納される。この、主記憶に格納されている命令 (といってもこれもビット列ですが) の並びを「プログラム」という (図 5)。こういうのを「プログラム内蔵方式」(stored program archtecture) と呼び、偉大な発明だとされている (なぜか分かりますか?)。

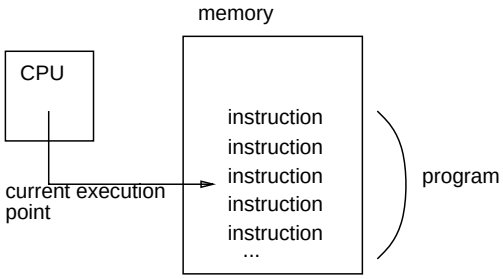

図 5: プログラム内蔵方式計算機

2.4 OS の位置づけ

では、もう一度質問しよう。

なぜそんなこと (つまり、あなたの欲するようなプログラムを
主記憶に入れて CPU に実行させること) が可能になるのか?

例えば次のような方法が考えられまる。

方法 1: 工場で、主記憶の一部にプログラムを書いて、消えないように固定しておく。

方法 2: 指令に従って、プログラムを主記憶に転送してきて実行するプログラムを走らせておく。

方法 1 は冷蔵庫とか洗濯機とかに組み込まれている小さい計算機 (制御用マイコン) ではよく使われる。このように一度書き込んだら消えないメモリのことを ROM(Read Only Memoly) という。しかし、それだとその計算機がやることは買ってきたときにもう決まってしまって、それ以上変更することは決してない。そうではなくて、買ってきたプログラムやあなたが作成したプログラムをそのつど動かしたいわけなので、方法 2 の方が一般的である。そして、その「プログラムを実行してくれるプログラム」を OS(operating system) と呼ぶ。もちろん、OS がやる仕事がこれだけということはないが、任意のプログラムを起動することは OS の主要な仕事の一つ。そうす

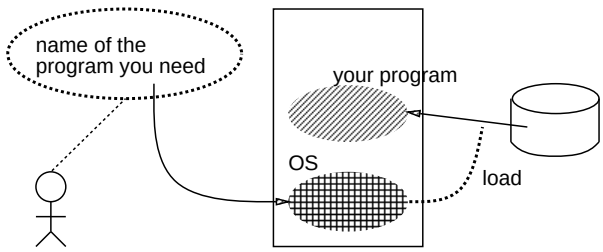

図 6: OS によるプログラムローディング

ると、あなたがある計算機を使う時は (OS をそうたびたび変更することはまずないですから) 図 7 に示すように、いつも同じ OS が動いていて、裸の計算機と OS が一体になったものがいつもあなたの目に触れていることになる。そしてその上で自分のプログラムや買ってきたプログラムを動かすわけである。

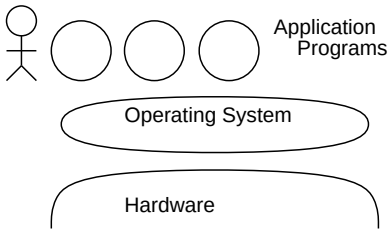

図 7: OS の位置づけ

3 マルチプログラミングとプロセス

3.1 マルチプログラミング

ところで、前節図 7 には「プログラム」を表す○が複数描いてあるが、これらは時間的には順番に実行されるものだと思いましたか?実は、現在の OS では普通、複数のプログラムを並行して、同時に、複数個走らせることができる (multiprogramming)。これがどうやって可能になっているかを図 1 に示そう。 つまり、主記憶には走らせたいプログラムを複数入れておき、そのうちの一つ (たとえばプログラム A) を実行開始する。ところで CPU には時計が備わっていて、一定時間実行すると一旦 A の実行を停止して OS のプログラム部分 (ここには描いていないが) の命令に切り替わり、OS が「A はとりあえず十分動いたから次は B の番だ) と判断すると今度はプログラム B の命令の実行に移る。再び時間がくると今度は B から C に移り、そしてまた C から A へ戻る。もちろん A へ戻った時はさっき実行した A の命令の次の命令から続きをやる。従ってマルチプログラミングでは本当に複数のプログラムが同時に動くのではなく、それぞれが小刻みに切り替わ

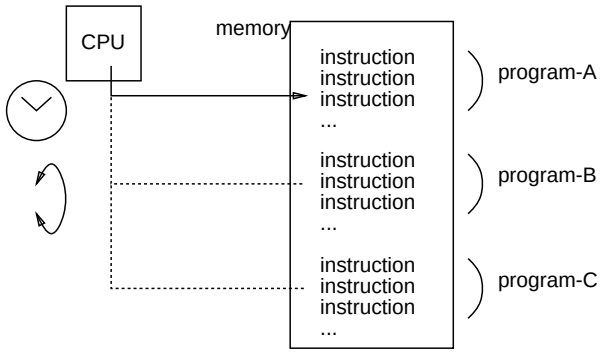

図 1: マルチプログラミングの実現

りながら順番に動いていると考えるのが正しい。しかし CPU は十分速いので同時に動いているように見える。

3.2 プロセス

この、「実行している状態のプログラム」のことを「プロセス」と呼ぶのが普通である (メーカーによって別の名前で呼ぶこともある)。だから計算機システムの中では目には見えないが沢山のプロセスが並行して走っている、というイメージを持っていただきたい。目には見えない、と書いたが実は Unix にはプロセスの状態を観察する指令 ps がある。その主要な使い方は次の通り。

```
ps x      -- 自分のプロセスをすべて表示する
ps ax     -- 他人のも含めすべてのプロセスを表示する
ps lax    -- 〃、ただしより詳しい表示
ps uax    -- 〃、ただし CPU 使用頻度順に、ユーザ名つきで表示
ps vax    -- 〃、ただしメモリ使用頻度順に表示
```

一般に「a」の指定してあるやつは a をとることで「自分のプロセスのみを表示」になる。普段は自分のプロセスだけ見れば十分なことが多い。次に、とあるマシンでの ps ax の例を示す。このように、Unix では特に自分が何もしてなくても多数のシステムプロセスが動いている。

```
  PID TT STAT TIME COMMAND
    0  ? D    2:08 swapper   <- システムプロセス群...
    1  ? I    0:03 init -
    2  ? D    0:37 pagedaemon
    3  ? D    0:11  (syncd)
   23  ? IW   0:03 portmap
   38  ? I    0:12 /etc/ypbind
   47  ? S   18:22 /etc/routed
   53  ? S    0:06 /etc/syslogd
   58  ? I    1:52 /etc/timed -t
   63  ? I    1:05 -accepting connections (sendmail)
   70  ? S    2:11  (biod)
   71  ? I    2:10  (biod)
   72  ? IW   2:13  (biod)
   73  ? IW   2:16  (biod)
   81 co IW   0:00 /etc/rpc.statd
   84 co IW   0:00 /etc/rpc.lockd
   91  ? IW   0:02 /etc/snmpd
  117  ? S<   9:45 /usr/etc/amd -p -r -d ...
  150  ? S    2:09 /etc/update
```

```
  153  ? S    6:41 /etc/cron
  158  ? I    0:09 /etc/inetd
  165  ? I    0:12 /etc/ewhod ksa
  173  ? IW   0:12 /usr/lib/lpd -default_domain is.titech.ac.jp
  179 co IW   0:00  (consd)
  188 co IW   0:00 - std.9600 console (getty)
  198  ? IW   0:00 rpc.mountd
 1325  ? IW   0:04 rpc.rquotad
 6566  ? I    2:02 /usr/bin/X11/xdm
12532  ? I    0:01 -kx09:0 (xdm)
20609  ? I    0:00 -kx07:0 (xdm)
24029  ? I    0:00 -kx10:0 (xdm)
24739  ? I    0:00 -kx06:0 (xdm)
24747  ? I    0:03 /usr/bin/X11/xterm -geom 78x18+0+0 -T 9:cons -n 9:cons -e /u
24763  ? I    0:00 /usr/bin/X11/twm
24764  ? S    0:00 /usr/bin/X11/xclock -u 1 -geom 100x100-0+0
24765  ? S    0:01 /usr/bin/X11/xterm -geom 80x24-0+102 -T 0:ksd -n 0:ksd -e /u
24898  ? I    0:00 -kx08:0 (xdm)
24766 p0 I    0:01 -usr/new/csh (csh)
24767 p1 S    0:00 -usr/new/csh (csh)
24918 p1 R    0:00 ps ax
```

ところで、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。

ps の表示には必ず PID(プロセス ID) という番号が含まれている。実はこれを使ってプロセスをいろいろ操作することができる。操作するには普通 kill という指令を使う。

`kill -STOP <PID>`	プロセスの実行を一時凍結する
`kill -CONT <PID>`	凍結したプロセスの実行を再開する
`kill -TERM <PID>`	プロセスに「終って欲しい」と信号する
`kill -KILL <PID>`	プロセスを強制終了させる

プロセスを凍結したり再開したりしてみるのは演習の内容としてはなかなか面白いが、普段はどちらかというとあとの 2 つを使っていらないプロセスを終らせることの方が多い。もちろん、自分が起動したプロセスでなければ勝手に止めたり終らせたりはできない。誰のプロセスかは ps で「u」オプションをつければ分かる。

ところで、ps というのは Unix にもとからある指令なのだが、この表示をもっと見やすいように改良した sps という指令が使えるシステムも多い。sps だといちいち指定しなくてもプロセスの所有者や親子関係が表示されるので分かりやすい。オプション指定についてはマニュアルページを見るべし。上の「ps ax」に対応する「sps a」を示しておこう。

```
Ty User     Proc# Command
   root         0 Unix Swapper
   root         1 init -
   root         2 Unix Pager
   root         3 ()
   root        23 portmap
```

```
   root        38 /etc/ypbind
   root        47 /etc/routed
   root        53 /etc/syslogd
   root        58 /etc/timed -t
   root        63 -accepting connections()
   root        70 ()
   root        71 ()
   root        72 ()
   root        73 ()
co root        81 /etc/rpc.statd
co root        84 /etc/rpc.lockd
   root        91 /etc/snmpd
   root       117 #amd -p -r -d is.titech.ac.jp -w 240 -l /var/adm/am.log -a /a
   root       150 /etc/update
   root       153 /etc/cron
   root       158 /etc/inetd
    |         198 rpc.mountd
    |        1325 rpc.rquotad
   root       165 /etc/ewhod ksa
   root       173 /usr/lib/lpd -default_domain is.titech.ac.jp
co.root       179 (consd)
co.root       188 - std.9600 console(getty)
   root      6566 /usr/bin/X11/xdm
    *       12532 -kx09:0()
    *       20609 -kx07:0()
    *       24029 -kx10:0()
    *       24739 -kx06:0()
     *      24747 #xterm -geom 78x18+0+0 -T 9:cons -n 9:cons -e /usr/new/csh
      |kuno 24763 /usr/bin/X11/twm
      |     24764 /usr/bin/X11/xclock -u 1 -geom 100x100-0+0
      |root 24765 #xterm -geom 80x24-0+102 -T 0:ksd -n 0:ksd -e /usr/new/csh
       *kuno24767 -usr/new/csh ()
        *   24938 sps a
      *     24766 -usr/new/csh ()
    *root   24898 -kx08:0()
40 (43600k) processes, 2 (576k) busy, 31 (6344k) loaded, 9 (160k) swapped
```

以下の例ではとりあえず純正の ps の方を使っているが、皆様は好みに応じてどちらを使っていただいてもよい。

3.3 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。例えば次はとある「ps l」の一部であるが:

```
      F UID   PID  PPID CP PRI NI  SZ  RSS WCHAN    STAT TT  TIME COMMAND
 408201  20 10702 10697  0  15  01424  112 pause    S    p0  0:02 -usr/new/csh
      1  20 10711 10702  8  27  0 136  344          R    p0  0:00 ps l
```

この中の PID と PPID(Parent PID) を見てみると、下のプロセスの親が上のプロセスになっている。つまり csh のプロセスが ps 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 csh というプログラムによって読みとられる。csh 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 3 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。

ところでさっきの後でもう一度 ps をやった結果が次のものである。

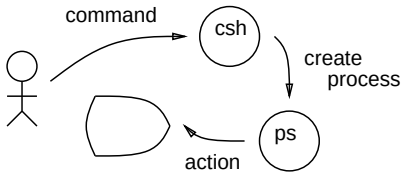

図 2: コマンドインタープリタ

```
     F UID   PID  PPID CP PRI NI  SZ  RSS WCHAN    STAT TT  TIME COMMAND
408201  20 10702 10697  0  15  01424  112 pause    S    p0  0:03 -usr/new/csh
     1  20 10719 10702  2  25  0 136  344          R    p0  0:00 ps l
```

これを見るとcshのPIDは同じだがpsの方は違っている。つまりさっきのpsはあれで一旦終ったので今度はまた新たにプロセスを作ったのだが、cshそのものは同じままなわけである。この様子を図3に示す。つまり、csh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cshによって作られる。このようなやり方にはどんな利点があると思うか?(実は、OSによっ

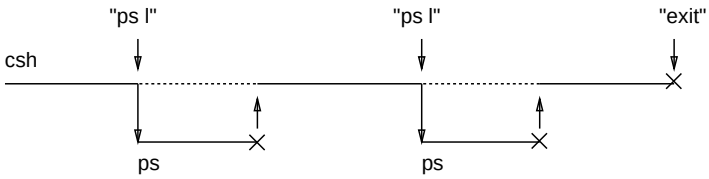

図 3: cshによるコマンドの発行と待ち合わせ

ては指令を起動するたびにコマンドインタープリタがあった場所に起動したいプログラムを読み込んで実行してしまい、それが終ったらまたコマンドインタープリタをそこへ読み込んできて次の指令を受け付ける、というやり方のものもある。それだとどういう困ったことが起きるか?)

- csh を 1 度起動すればいいので初期設定の手間が気にならない
- 様々な情報を csh の中に保持しておくことで使いやすくできる
- csh と各プログラムの間での思わぬ干渉をなくすことができる

cshが終るのは、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csh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、csh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。例えば、「sleep 30」(これは何もせずただ30秒待つ、という指令である)という指令を実行して、その完了を待たずにpsもやる、というのを試してみる。

```
% sleep 30 &
[1] 10720
% ps l
     F UID   PID  PPID CP PRI NI  SZ  RSS WCHAN    STAT TT  TIME COMMAND
 408201  20 10702 10697  0  15  01424  112 pause    S    p0  0:03 -usr/new/csh
   8201  20 10720 10702  6  15  0  24  184 pause    S    p0  0:00 sleep 30
      1  20 10721 10702 16  29  0 136  352          R    p0  0:00 ps l
%
[1]    Done                  sleep 30
```

確かに、sleep と ps が同じ csh の子供になっている。この様子を図 4 に示す。こういうことが簡単にできるのも Unix 方式の利点である。

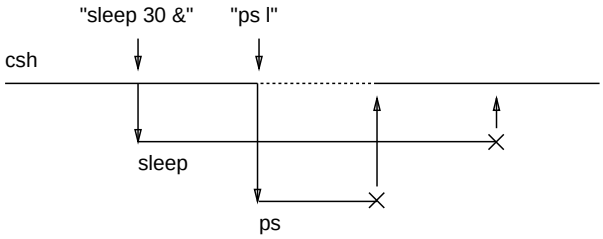

図 4: &つきのコマンドの発行

A 付録

A.1 このレクチャーの目的、運用など

このレクチャーの目的は皆様にとっては「単位を取る」ことかと思うが、こちらにとっては皆様に「前述の、計算機環境の背後にある体系について、理解し使いこなせるように」なっていただくことであろう。うんうんいって 2 時間も掛かるような仕事がほんの 5 分で済ませられる、ということが今後一生の間に 20 回ほどあるなら (保証はしませんが)、このレクチャーでつぶした時間くらい安いものである。この目的にちょうどぴったし合う教科書はないようなので教科書は指定しない。

参考書:カーニハン他、Unix プログラミング環境、ASCII

を指定してあるが、毎回持ってくる必要はない。でもよくできた本なので、疑問な点をひもといてみたり、ネタが必要な時に読んでみたり、暇なときめくってみるようお勧めする。

さて、次に皆様の主目的である「単位」の方であるが、次のような方式で採点する。そもそもこういう種類の話はレクチャーだけ聞いても絶対身につかない。実際に計算機で体験することが肝要である。だから、講義の時間は短めに終るよう努力するので、残った時間にいくばくか皆様の自由時間を追加していただいて、毎回最後に列挙してある「練習問題」を実地にやって下さるようお願いする。そして全 14 回 (といっても何回かは休講もあると思うがそれも実地演習の時間ね) のうち 5 回 (これは各自選択してよい) 以上に対して「報告」を提出していただき、それに基づいて成績をつける。(出席用紙を配布して出席を取ることもあるが、その出席点も多少考慮される。) [1]

「報告」であるが、これはいわゆる「レポート」というほどがんばらなくても結構である (でないと 5 回も出せませんから)。要は、練習問題を実際に自分でやったという証拠 (リスティングとか) と、何をやったかという解説、そしてそれに対する「スルドイ」コメントをお願いする。形式は次の通りとする。

- 必ず、A4 版の用紙を使用し、ばらばらにならないよう綴じる。
- 1 枚目は表紙とし、学科、氏名、学籍番号、提出日付、問題番号を記す。

[1] ところで、「演習」を行なうには当然 Unix マシンが必要である。もちろん情報科学科の皆様にはこれは問題ないが、それ以外の学科の方でもバークレー版 Unix が何らかの形で利用可能であればそれを使ってやっていただければ構わない。おなじ Unix でも SystemV だとちょっと違う所があるが、まあ何とかなるだろう。全く Unix が触れないという人は、お気の毒であるがセンターのアカウントが取れるようになってからいらっしゃるのが結局は早道ですよ、と申し上げるほかない。

- なかみは上記の趣旨にあう限りにおいて自由である。
- 必ず、コメントを記す。コメントに期待されるものはあなたがこの問題をやった結果どう思ったか、どんなことが分かったか、といった内容だと思う。(「面白かった」なんて抽象的なのはだめよ。)

A.2 1回目の演習

本日の演習は、Unix マシンに login して今日学んだようなプロセスの挙動を確認し、また直接は説明を聞いていないような様々なプロセスの挙動を観察してみることです。つぎのようなテーマをやってみてください。

1. 自分が Unix を使っている状況の ps を表示させ、どのプロセスがどんな役割であるかを類推してみよ。その類推が正しいかどうかを確認するため、適宜プロセスを凍結してみて、確かに予想された機能が停止するかどうかを調べてみること。[2]

2. あるマシンに誰かが login して、いくつか指令を実行して、終了するまでのプロセスの移り変わりの状況を観察して図に描いてみよ。これをやるにはあるマシンに入った状態で、そのマシンにつながっている別の端末から誰かに login してもらいながら観察すればよい。もちろん一人二役してもよい。

3. 次のような場合にはプロセスの状況はどうなっているか観察してみよ。(1)nemacs でメールを書いている状態、および他人から来たメールを読んだ後、返事のために nemacs を動かしている状態。(2)vn そのほかのニュースリーダでニュースを読んだ後、フォローの記事を nemacs で書いている状態。いずれもその瞬間だけでなくその前後の過程も追ってほしい。[3]

4. 3 つ窓を開きそれぞれで「sleep 30 &」をやった状況と 1 つの窓で「sleep 30 &」を 3 回やった状況の違いは何か、実地に観察してみよ。プロセスの親子関係をよく調べること。

5. 「sleep 30 &」を実行したあと、そのプロセスを強制的に終了させて見よ。あるいは、凍結して、30 秒たっても終らないことを確認してから解凍してみよ。

6. X-Window を動かした状態で「xclock -a -u 1 &」とやると秒針つきの時計が表示される。これを凍結したり解凍したりしてみて秒針が止まることを確認せよ。凍結した分だけ遅れるかどうか?それはなぜか?

7. 友達と二人で隣どうしのマシンに login し、交替で一つずつ自分のプロセスを殺していき、どちらが長く生き残るかというゲームをやって報告せよ。もちろん、始める時点でどう設定しておくかが鍵である。[4]

A.3 1回目の課題

本日の課題は、上記 9 項目のうち最低 2 つ以上を選び、それをやってみた結果について報告することとします。

[2]窓 1 個だけでやるとその窓から打ち込めなくしてしまった時アウトだから、窓を複数個開いてやること。

[3]メールを書いたことがないとかニュースを読んだことがないという人にはできない課題ですが。この際それくらいマスターしてほしいなあ。

[4]やる気になればすごくこすい方法も取れる (分かります?)。互いにどうゲームのルールを取り決めるかにもよるが。