計算機科学基礎'13 ♯ 2 – システムソフトウェア

久野　靖*

2013.4.17

今回は「ソフトウェア」の前半として、私たちがコンピュータを使う時に常にお世話になっている「システムソフトウェア」について、その概念、機能、原理などを学んで行きます。これらを知ることで、コンピュータシステムが「なぜそういう風に動くのか」が分かるようになると思います。

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システムソフト

あなたがふだんソフトを使っているとき (たとえばブラウザで Web を見ているとします)、コンピュータ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 (たとえばワープロソフトなどに) 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** (application software) — ユーザが各々の仕事を実行するためのソフト。
- **システムソフト** (system software) — 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システム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** (operating systems、OS) — アプリケーションが動く下ざさえとなる機能を提供するひとまとまりのソフトウェア。
- **ミドルウェア** (middleware) — データベース管理システム、Web サーバなど、(OS と同様の意味で) アプリケーションが動く基盤となるが、OS とは独立に開発・提供されているもの。
- **言語処理系** (language processors) — プログラムを記述し動かすための機能を提供するソフトウェア。次回に取り上げる。
- **ユティリティ** (utilities) — 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。

人によっては、これらの分類には多少変動があるかも知れません。今回は Unix を題材として、まず OS について、続いてユティリティについて扱います。ミドルウェアについてはデータベース、Web サーバなど個々の話題の中で (別の回に) それぞれ取り上げます。

1.2　OS の各種の役割り

上で OS の役割りは「アプリケーションが動く下ざさえ」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。

*経営システム科学専攻

既に見たように、コンピュータのハードウェア命令は、メモリとレジスタの間のデータを転送や四則演算など、ごく基本的な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、文字列を読み込んだり表示するなどの機能を実現するのには、かなり長いコードを必要とします。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、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、以下は OS の重要な役割りだと言えます (図 1)。

- 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

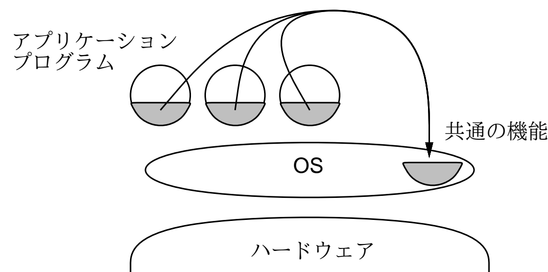

図 1: OS による標準的機能の提供

次に、現在のコンピュータ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** (multitask) 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 Web を見たり、といった具合です。従って、次のことも OS の役割りです (図 2)。

- 複数のプログラムが並行して動作するのを管理する

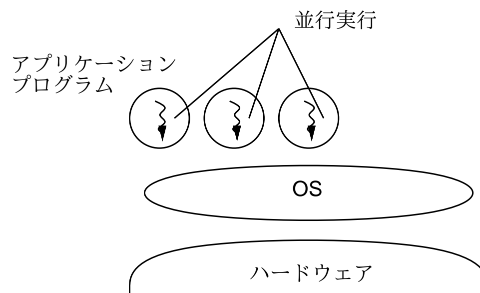

図 2: OS によるマルチタスク機能の提供

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。つまり、以下のことも OS の基本的な役割りなのです (図 3)。

- 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

さて、そうやって並行動作している複数のプログラムが同じメモリ番地やディスク上の領域を使おうとしたら、やはり混乱が起きます。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りますね? ですから、以下のことがらも OS の重要な仕事です。

- コンピュータ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 (読み/書き) を管理する

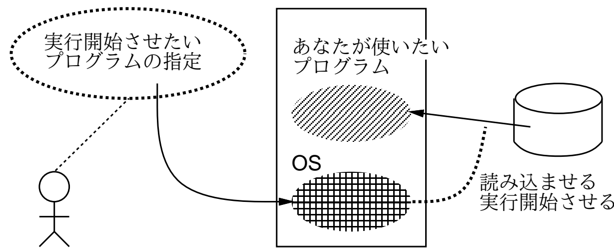

図 3: OS によるプログラムの実行開始

ところで見かたを変えると、コンピュータ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** (resource) だと言えます。ですから、OS の主要な機能は要するに次のようにまとめることができます。

- コンピュータ内部の各種資源を管理する

あなたが動いているコンピュータを目にする時、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ます。そして、あなたやあなたのプログラムがコンピュータを使おうとする時、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)。

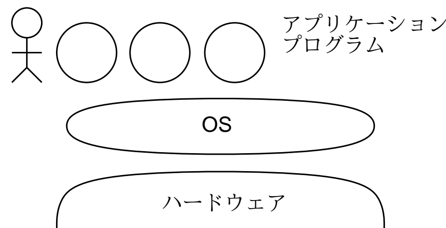

図 4: OS とユーザ、アプリケーションの関係

1.3 マルチタスクとプロセス

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、その中でも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から見てみましょう。そもそも、どのようにしてそのようなことが可能になるのだと思いますか?

まず、コンピュータの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)。

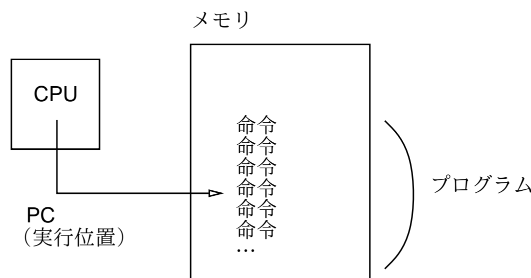

図 5: CPU による命令の実行

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。そして、CPU についている**タイマー** (timer) と呼ばれる機能を動かした状態でまずプログラム 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。

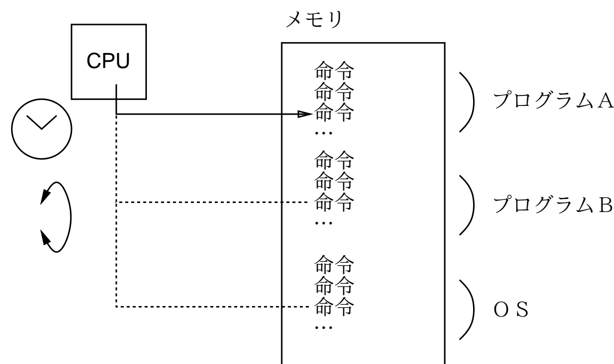

図 6: マルチタスク機能の実現

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。一般にこの「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。

CPU を複数搭載したシステム (マルチプロセサ、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、コンピュータ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]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。

1.4 ps — Unix でのプロセス観察

Unix では、**ps**(process status) コマンドによって、現在動作中のプロセスを観察できます (指定す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

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。[4]

```
  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)
  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**端末番号**が記されているプロセスが、ユーザが直接使っているものです。それ以外の大部分のプロセスは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 では各ユーザが自分のために複数のプロセスを利用するのに加え、システム自体の運用のためにも多数のシステムプロセスが動作しています。

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
```

[3]Unix システムの系統ごとに ps のパラメタ指定は違っています。ここでは FreeBSD の場合を説明しています。

[4]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。

```
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。[5]

ところで、ps の表示にある TIME というのは「そのプロセスがこれまでに消費した CPU の時間」を表しています。これを確認するため、C 言語で 1 行だけのプログラムを書いて動かしてみましょう。

```
% echo 'main() { double i = 10e9; while(--i); }' >test.c ←C 言語プログラム
% gcc test.c    ←実行形式に変換
% a.out &       ←&つきで実行開始
[1] 59412
% ps
   PID  TT  STAT      TIME COMMAND
 96644  p0  S      0:00.02 bash
 96656  p0  R      0:01.22 a.out    ←現在 1.22 秒 CPU 消費
 96658  p0  R+     0:00.00 ps
% ps
   PID  TT  STAT      TIME COMMAND
 96644  p0  S      0:00.02 bash
 96656  p0  R      0:03.98 a.out    ←現在 3.98 秒 CPU 消費
 96658  p0  R+     0:00.00 ps
%
[1]+ Done  a.out
```

[5] kterm のプロセスが表示されていませんが、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ため、「自分のプロセスを表示」に含まれません。全部のプロセスを表示させれば見えます。

```
%
```

このプログラムは、(実数型の) 変数 i に 10.0×10^9 を入れ、それを 1 ずつ減らしながらループし、0 になったら終了します。それを test.c というファイルに入れ、gcc コマンドで実行形式に変換し、「&」つきで実行開始します。その後 ps で複数回観察すると、CPU 使用時間が増えていく様子が分かるわけです (だいたい 10 秒くらいで終わります)。

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。[6]

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 bash のプロセスになっています。つまり、bash が ps その他のプロセスを生成しているわけです。

これは何を意味するのでしょう? ユーザがキーボードからコマンドを打ち込むと、それは bash によって読み取られます。bash はその入力を見て、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ます (具体的には、OS に依頼してプロセスを生成します)。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文字列を受け取り、その内容に応じた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** (command interpreter) と呼びます (Unix ではコマンドインタプリタ自体も普通のプログラムであり、プログラムの起動などは OS に頼むことで実現しています)。

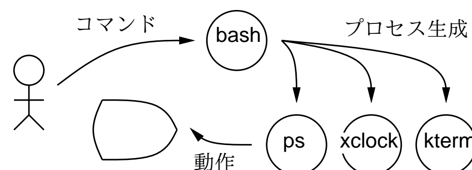

図 7: コマンドインタプリタ

[6]「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。

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。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。[7]

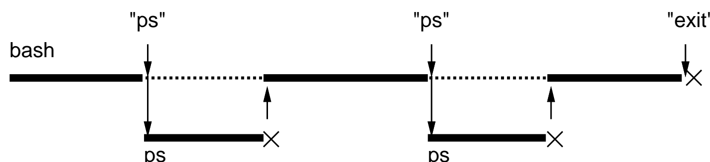

図 8: bash によるコマンドの発行と待ち合わせ

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。そういうときはコマンドの最後に「&」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」という指定ができます。先に 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。

演習 2-1 「xclock -analog -update 1 &」により秒針つきの時計を作り、この時計の PID を調べて、kill コマンドにより凍結/再開の操作ができることを確認しなさい。確認できたら、以下のテーマから 1 つ以上 (できれば全部) 選び、探究してみなさい。

a. Emacs エディタを動かし、同様に凍結/再開できることを確認する。凍結している間に窓に対してキーを打った場合どうなるかを観察する。なぜその観察結果が起きているのか、理由を推測すること。できれば、その推測を確認する実験をさらに考えて試し、結果を報告できるとなおよい。

b. console の窓で twm というプロセスを探し、このプロセスを凍結/再開したときにどのようなことが起きるかを観察する。なぜその観察結果が起きているのか、理由を推測すること。できれば、その推測を確認する実験をさらに考えて試し、結果を報告できるとなおよい。

c . ブラウザ (Mozilla) を動かした状態で console の窓でプロセスを観察し、ブラウザのプロセス群を探しなさい。その中で最も CPU 時間を多く使っているものを凍結した状態で窓の上に別の窓を重ね、起きることを観察し報告しなさい。できれば、なぜそうになるのかを考察し、それを確認する実験を行って結果を報告できるとなおよい。

「◎」の条件: (1) 小問を 2 問以上解答しており、(2) いずれも「できれば」の部分までやってあって、(3) まともな (と担当が考える) 考察が書いてあること。

演習 2-2 プロセスの観察に関する以下の課題から 1 つ上 (できれば全部) 選び、探究してみなさい。

a. 本文にある「CPU を 10 秒くらい消費するプログラム」の別版として、次の C 言語プログラムを考えてみます (OS の待ち機能を使い、CPU を消費せずに待ちます)。

[7] 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。

```
main() { sleep(10); }
```

このプログラムも同様にして動かしてみて、先のプログラムかこのプログラムかを教えられていない場合でも、ps の表示からどちらであるか判定する方法を考案しなさい。できれば、「CPU 消費」「CPU 非消費」の 2 モードを切り替える (10 秒間 CPU を消費し、10 秒間待ち、また 10 秒間 CPU を消費する) プログラムについて、現在どちらのモードにいるかを調べるのもやってみるとなおよいです。

b. 「`ps lx`」を用いてプロセスの親子関係を観察できることを確認し、自分が実行したコマンドの親子関係を整理して述べなさい。できれば、「親のプロセスが終了した子供プロセスの親はどうなるか」という疑問に答える実験を実施し、結果を報告するとなおよいです。

c. やはり「`ps lx`」を用いて、さまざまなやり方で (コマンドを打ち込んだりメニューを使ったりして) コマンドを起動したとき、その親プロセスがそれぞれ何になっているのかを観察して報告しなさい。できれば、親のさらに親までだとって行き、自分のプロセス全体がどうなっているのかも観察して報告できるとなおよい (そのためには「`ps lax`」も使う必要があるでしょう)。

「◎」の条件: (1) 小問を 2 問以上解答しており、(2) いずれも「できれば」の部分までやってあって、(3) まともな (と担当が考える) 考察が書いてあること。

2 2 次記憶とファイルシステム

2.1 2 次記憶の役割

メモリは主記憶装置ともいわれるように、コンピュータシステムにおける「主要な」記憶装置であり、プログラムが動作する上で不可欠ですが、その容量は限られているうえに、電源を切ると記憶内容は消えてしまいます。しかし、今日のコンピュータシステムには重要なデータを大量に蓄積する必要があり、それらのデータは電源が切られても (ソフトのトラブルでシステムを再起動する場合もこれに相当します)、内容が損なわれることは許されません。このためには **2 次記憶** (secondary storage) 装置が使用されます。2 次記憶に必要とされる条件には以下のものがあります。

- 安定記憶 — 電源を切っても消えず、内容が勝手に書き変わったりしない。
- 容量/コスト — 大量のデータを格納できる。1 ビットあたりのコストが低い。

これらの条件を満たし、今日のコンピュータシステムで広く使われている 2 次記憶装置としては**磁気ディスク装置** (hard disk drive、HDD) が代表的です。このほか、電源を切っても内容が保持されるような半導体メモリである**フラッシュメモリ** (flush memory) もモバイル機器を中心に使われていますが、ビット当たり単価はやや高めです。ただし高速なので、通常の PC でもこれを磁気ディスク装置の代替として使用することも増えています。これを **SSD**(solid state disk) と呼びます。[8][9]

磁気ディスク装置の原理は図 9 のようなもので、回転する円盤 (**プラッタ**) の表面に磁気コーティングがしてあり、読み書きヘッドを用いて同心円状に情報を記録します (1 つの同心円をトラックと言います)。1 つのトラックには一定サイズ (具体的には 512 バイト) のかたまり (**セクタ**) 単位で情報を記録します。ヘッドは半径方向に動くので、これによって多数のトラックを選択します。PC などに入っているディスクではプラッタ数 1、ヘッド数 2(つまり両面ぶん) のものが主流ですが、これでも数百 GB(ギガバイト、10^9 バイト)〜数 TB(テラバイト、10^{12} バイト) の容量を持ちます。[10]

[8] また、フラッシュメモリは書き換え回数の上限が数千回〜数万回程度で、それを超えると壊れてしまうので、実質何回でも書き換え可能なディスクの代わりに使うためには書き換え回数を押える工夫が不可欠になります。

[9] これらのほかに、**リムーバブル** (取り外し可能) なストレージとして、CD、DVD、USB メモリなど多様なものがありますが、これらはデータ交換が主眼で、読み書き速度や容量の点では不利です (最近はだいぶ改善されましたが)。

[10] 磁気コーティングした円盤の代りに光磁気反応素材を用いたのが (MD などでも使っている)「光磁気ディスク」、製造時に固定したパターンを付けておきレーザー光で読み出すのが「CD-ROM」や「DVD-ROM」。これらのパターンをレーザー光で書き換えられるようにしたものが「CD-R」「DVD-R」「DVD-RW」「DVD-RAM」など。これらはいずれも磁気ディスクに比べると読み書き速度が遅い。

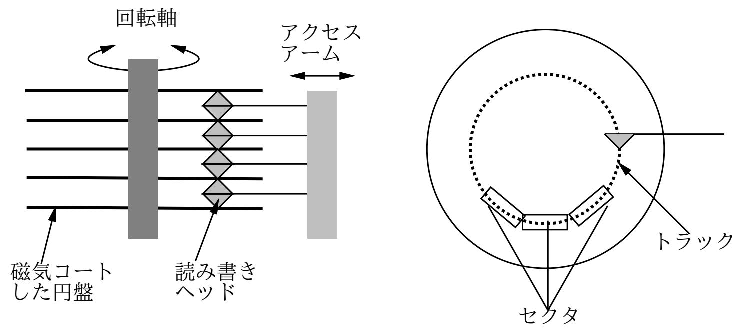

図 9: 磁気ディスク装置の構造

たとえば、2TB で 1 万円 (やや高め) のドライブでは、1 円あたり 200MB(メガバイト、10^6 バイト) という計算になります。USB メモリ (32GB で 2 千円くらい?) と比べるといかに廉価か分かると思います (しかもめったに壊れません)。

では、磁気ディスク装置の弱点は何でしょうか? それは「読み書きが遅い (正確には読み始める/書き始めるまでが遅い)」ことです。これは、回転している円盤の上に、ヘッドを移動させて位置決めして記録することによるもので、このような機械的な動きがあるとどうしても時間が掛かります。たとえば、平均的な読み書き時間は典型的なディスクで数十 msec〜数百 msec くらいになります。これは、主記憶装置の読み書き時間が数十 nsec 程度なのと比較すると、「百万倍〜1 千万倍 (!!) 以上」も遅いことになります。このことは、ブロック単位でのみ読み書きができることと併せて、2 次記憶の利用のしかたに対して大きな影響を与えます。[11]

2.2 2 次記憶の速度を測る

実際に 2 次記憶のアクセス速度を測ってみましょう。ここでは Unix の `dd` というコマンドを使います。このコマンドは、データ転送が本来の用途ですが、これで「無限にあるゼロを、指定したバイト数だけファイルに転送する」所要時間を測ることで、2 次記憶の書き込み速度が調べられます。

```
% time dd if=/dev/zero of=/var/tmp/t bs=4096 count=10000
10000+0 records in
10000+0 records out
40960000 bytes transferred in 1.458445 secs (28084705 bytes/sec)
real    0m1.472s
user    0m0.001s
sys     0m0.215s
% rm /var/tmp/t  ←作ったいらないファイルはすぐ消す
%
```

上の例では入力ファイル (`if`) が「無限のゼロ」(`/dev/zero`)、出力ファイル (`of`) が`/var/tmp/t`、1 回に読み書きするブロックのサイズが 4096 バイト、転送するブロック数が 10000 個になります。ということは、転送量が 40M バイト、所要時間が (今回は real つまり実際の所要時間で見ます)1.5 秒ですから、およそ 27M バイト/ 秒の書き込み速度ということになります。しかし、`/var/tmp/t` とは何のファイルでしょうか? 実は我々の環境では、`/var/tmp/`はそれぞれのマシンに搭載されたディスク装置上の作業場所に相当し、ここに書くことでディスクの速度が測れます。普通に (自分の) ファイルを作るとどう違うか、試してみましょう。

```
% time dd if=/dev/zero of=t bs=4096 count=10000
```

[11] フラッシュメモリを使った 2 次記憶は動く部分がないのでハードディスクよりも高速ですが、メモリの構造や伝送路の長さなどの関係があり、やはり主記憶よりはずっと低速です。

```
10000+0 records in
10000+0 records out
40960000 bytes transferred in 4.316991 secs (9488090 bytes/sec)
real    0m4.346s
user    0m0.009s
sys     0m0.238s
% rm t     ←作ったいらないファイルはすぐ消す
%
```

ずっと遅いですね? これは、自分のファイルはファイルサーバ上に保管されるので、ネットワーク経由でサーバまで送り、そこのディスクに格納するのに掛かる時間になるためです (ということは、サーバ sma の上で同様にして測ると、今度は速いはずです)。また、`/tmp/t` というファイルで試すと、`/tmp` はメモリ上にファイルを保管するので、ぐっと速くなるはずです。

演習 2-3 上記と同じ dd コマンドを使って、次の 3 通りの場合について所要時間を計測しなさい。(1) `/var/tmp/t` — 自分の使っているマシンについているディスクに書く。(2) `t` —自分のホームに (ファイルサーバ経由で) 書く。(3) `/tmp/t` —メモリファイルに書く (いずれも計測したらすぐ不要なファイルは消すこと)。これを、以下の 1 箇所以上 (できれば 3 箇所) で行いなさい。

a. console の窓で実行 — 自分が座っているマシンでの結果。
b. sma の窓で実行 — ファイルサーバマシンでの結果。
c. その他のサーバマシン (smb、utogw) でもやってみる。

計測条件 (どのマシンか) や計測値を明記し、計測から分かったことを整理して記すこと。

「◎」の条件: (1)2 箇所以上で計測を行っており、(2) まともな (と担当が考える) 考察が書かれていること。

2.3 ファイルとファイルシステム

磁気ディスクなどの 2 次記憶装置は、CPU 側からは「何番のセクタを読む/書く」という形で使います。しかしそのような裸の記憶装置を「そのまま」使えと言われたら困ると思いませんか。

- 自分はどのセクタにデータを保管すればいいのか分からない。
- 間違って他人のセクタにデータを書いてしまったら困る。
- 逆に他人に秘密の情報を勝手に読まれてしまったら困る。
- そもそも自分がどこからどこまでのセクタに何を入れたか管理するのが大変。
- 途中でデータの量が増えたらどうするか (すぐ後が空いていればいいが、既に使われていると続きはどこか「飛び地」に入れることに…)。

しかし実際には、我々がデータを保管する時には次のようなことが可能になっています。

- 保管が必要なデータを書き込む時に、自動的に必要なだけの領域が確保されてそこにデータが書かれる。追加も自由。
- もちろん、他人が勝手にその領域を読んだり書いたりはできないように保護される (必要なら読める/書けるように設定も可能)。
- そのデータには「名前」がつけられ、名前を指定することでいつでもそのデータを参照できる。
- データの「数」が沢山になったら、それをグループに分けて整理することもできる。

この、データ (バイトの列) を入れておく、名前のついた「いれもの」のことを一般に**ファイル** (file) と呼びます。ディスク装置は上で見たように単なるセクタの集まりですが、そののっぺらぼうの上に作られた「名前のついた」「きちんと大きさや場所の管理された」いれものがファイルなわけです。

これはちょうど先に学んだ、のっぺらぼうのメモリと CPU の上に多数のプロセスが作られるのと同様なのです。つまり、ディスク装置など裸の 2 次記憶装置上にコンピュータによって作り出された、**仮想化**された使いやすい 2 次記憶がファイルだということになります。

そして、OS のうちファイルを作り出す機能の部分を**ファイルシステム** (filesystem) と呼びます。ファイルシステムは、OS のうちでもディスク上の領域という資源を管理する部分だ、と考えることもできます。ファイルシステムは大きく分けて、データを格納するための領域を管理する**領域配置** (storage layout) 層と、個々のいれもの (ファイル) に対してそれは誰のもので誰はアクセスしていい、などの属性を管理する**属性管理** (attribute handler) 層から成っています。ここでは Unix を例に、両者をそれぞれ簡単に見てみます。

2.4 Unix の領域管理

ここでは、Unix のファイルシステムを例に 2 次記憶の領域がどう管理されているかを見て行きます。Unix では伝統的に、1 つのファイルごとに管理用データを **i ノード**と呼ばれる構造に格納します。また、ファイルシステムが扱う読み書きの単位を**ブロック** (block) と言い、通常はセクタを 2〜8 個程度集めたもの (1024 バイト、2048 バイト、4096 バイトなど) となっています。

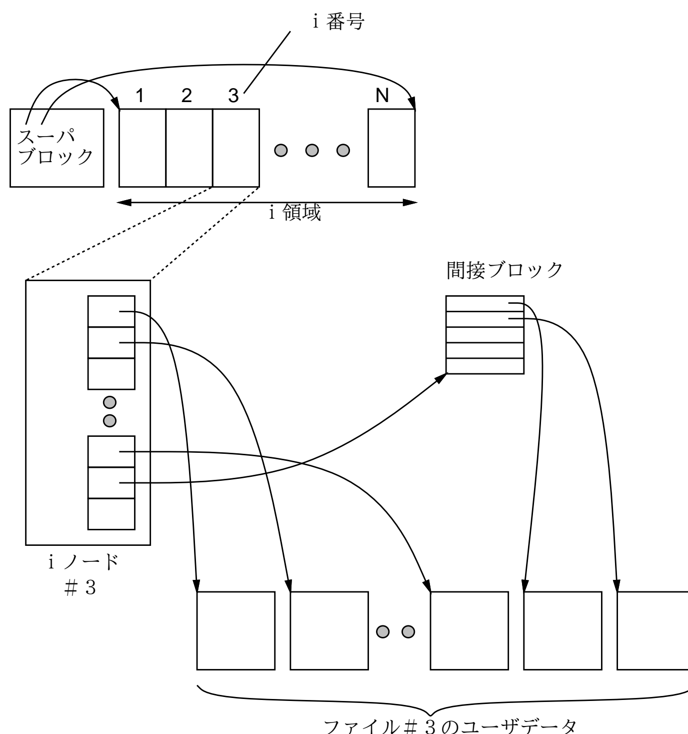

図 10: i ノードとデータブロック

i ノードの領域はディスクの先頭付近に一定量取られています。そして**スーパブロック** (super block) と呼ばれる特別なブロックがディスクの先頭にあり、ブロックサイズ、i ノード領域の大きさ、未使用領域の情報などが格納されています。i ノードが先頭にあるので、「何番目」かを指定すれば計算によりその i ノードの位置を求めて読み書きできます。この番号を **i 番号**と呼びます。つまり、Unix の領域配置層では i 番号がファイルの「名前」なわけです (図 10)。

個々の i ノードには、属性管理層が扱う所有者や保護設定などのデータに加えて、ブロック番号を格納する場所が 12 個用意されています。そのうちの 10 個までは、そのファイルのユーザデータを格納するブロック番号を直接格納します。そのため、ブロックサイズが 4096 バイトであれば、40960 バ

イトまでのファイルならiノードからただちに「何バイト目はどのブロック」という情報が得られ、そこに対するディスクの読み書きが行えます。

ではそれより大きいファイルについてはどうするかというと、11番目の場所には「ブロック番号を格納するブロック (間接ブロック)」の番号を入れます。ブロック番号が4バイトだとすると、1つのブロックには1024ブロック分の番号が入れられるので、これで $(10 + 1024) \times 4096$ バイトまでのファイルが扱えます。その次はもちろん「最大1024個までの間接ブロックのブロック番号を格納するブロック (二重間接ブロック)」を入れ、ここまでで $(10 + 1024 + 1024 \times 1024) \times 4096$ バイトまでのファイルが扱えるわけです。うまくできているでしょう? [12]

2.5 属性管理層とその機能

2.5.1 名前

先に説明したように、属性管理層はファイルに付随するさまざまな属性を管理するのが仕事です。以下で主要な属性やその操作方法を、Unixを題材に見て行きましょう (なぜUnix前提かというと、属性管理層の具体的な性質はOSごとに異っているからです)。

ではまず名前から見て行きましょう。自分が持っている**ファイルの名前**を調べるには、`ls` コマンドを使います。ただの `ls` では名前の最初が「`.`」で始まるものは表示されませんから、それらもあわせて表示させたい場合には、`-a` というオプションを指定します。

- `ls` — 今いるディレクトリ (後述) にあるファイルの一覧を表示
- `ls -a` — 「`.`」で始まるファイルも含めて表示

たとえば次のようになります。

```
% ls
Calendar        Mail            t2              t4
GSSMSETUP       kanji1.txt      t3              test1.txt
% ls -a
.               .login          .mnews_setup    .xinitrc        t3
..              .logout         .mozilla        Calendar        t4
.bash_profile   .mailcap        .newsrc         GSSMSETUP       test1.txt
.bashrc         .mailrc         .profile        Mail
.canna32        .mh_profile     .twmrc          kanji1.txt
.emacs          .mime.types     .x11defaults    t2
%
```

「`.`」で始まるファイルが多数ありますが、Unixでは各種のプログラムごとに、固有のオプション設定などを「`.`」で始まるファイルに書く、という習慣になっているためです。そして、いつもそれらが表示されているとうるさいので、`-a` を指定しない限り `ls` はそれらを表示しないわけです。

2.5.2 さまざまな属性

ファイルには名前に加えて、さまざまな属性がついています。それらの属性を表示するには、`ls` にもっと別のオプションを指定します。

- `ls -l` — 長さを始めとする詳しい情報を表示。

たとえば次のような内容を格納したファイル `test.txt` があるものとしましょう。

```
% cat test1.txt
This is a pen.
That is a dog.
%
```

[12] 初期のUnixでは本当にこのような素朴な配置を使っていましたが、それでは1つのファイルを読むのにディスクのあちこちにアクセスしなければならず、速度が遅いことが分かって来ました。そこで、現在のUnixではもっとさまざまに工夫された構造を使うようになっています。ここでは一番分かりやすいので、最も初期の構造を説明しまています。

そしてこれらの英字等はそれぞれ 1 バイトで格納できるものとします (実際できます)。そうすると、このファイルの長さはいくつになるでしょうか? (答えを見る前に数えてみること!)

```
% ls -l test1.txt
-rw-------  1 someone  other  30 Mar 27 13:59 test1.txt
%
```

答えは「30 バイト」ですが、合っていましたか? まず、「空白」も文字として数えることは分かります。しかし空白を含めても、まだ目に見える文字の総数より 2 バイト多いですが、これは各の行の終りに**改行文字** (行の切れ目を表す文字) がくっついているためです。

さて、その長さの右にあるのは、ファイルを最後に変更した日時です。変更した日時は記録されていないと、都合が悪そうですね? (たとえば最近変更したファイルだけ見たいなどの場合に。)

一方、長さの左にあるのは、そのファイルの「持ち主」です。持ち主の情報はなぜ必要なのでしょう? それはもちろん、ファイルを持ち主だけが読めたりするように「保護」するためですね。というわけで、その説明に進みますが、その前に「グループ」にも触れておきます。Unix ではユーザの他にグループという概念があり、「一緒に作業する仲間」を表すのに使うことが想定されています。ファイルにもその所属グループの情報がくっついていて、次のコマンドで見ることができます。

- `ls -lg` — `-l` と同じだが、所属グループも表示

たとえば私の場合で見てみましょう。

```
% ls -lg t1              ↓久野は教員グループ
-rw-r--r--  1 kuno  faculty  15 Mar 27 14:11 t1
%
```

Unix では、ファイルの保護設定を**モード** (mode) と呼び、各ファイルごとに

User (持ち主) | Group (グループメンバー) | Other (その他の人)

それぞれについて、

Read (読める) | Write (書ける) | eXecute (実行できる)

か否かを各々設定できます。モード情報は `ls -l` の表示の最初の部分に含まれます。

```
-rw-------  1 someone  other  30 Mar 27 13:59 test1.txt
```

これは先の例ですが、持ち主 (`someone`) は読み、書きは可能だが実行は不可、それ以外の人にはどれも不可という設定を意味しています。実行というのは、実行可能な (マシン語のプログラムの入った) ファイルに対してのものなので、C 言語で「hello.」と打ち出すプログラムを作って試します。

```
% echo 'main() { puts("hello."); }' >test.c    ←1 行の C プログラム
% gcc test.c                                   ←コンパイル
% ls -l a.out           ←a.out というファイルに実行形式ができる
-rwx------  1 someone  other  4718 Mar 27 14:16 a.out  ←「X」つき
% a.out                 ←ファイル名を言うと実行
hello                   ←プログラムの出力
%
```

Unix ではコンパイラは「`a.out`」という決まった名前の実行形式ファイルを作るので、これのモードを見ると確かに実行可能になっています。今度はまた別のファイルを見て見ます。

```
% ls -lg t1
-rw-r--r--  1 kuno  faculty  15 Mar 27 14:11 t1
```

ファイル t1 は、持ち主 (kuno) は読み書きともに可能ですが、グループ faculty の人、および他の人には読むことのみ可能です。モードの変更は、**chmod** コマンドで次のような形で指定します。

- chmod 対象 (+|-) 許可 ファイル … — モードを設定する

「対象」は u、g、o どれか 1 つ以上 (それぞれ持ち主、グループメンバ、その他の人を表す)、「許可」は r、w、x どれか 1 つ以上 (それぞれ読み、書き、実行を表す) で、「+」はその許可を出し、「-」はその許可を取り除きます。たとえばさっきのファイルでやってみましょう。

```
% chmod ugo-rwx t1      ←全員に対して「R」「W」「X」を OFF
% ls -lg t1
----------  1 kuno  faculty  15 Mar 27 14:11 t1
% chmod u+rx t1         ←自分に対して「R」「X」を ON
% ls -lg t1
-r-x------  1 kuno  faculty  15 Mar 27 14:11 t1
% chmod go+x t1         ←グループ/他人に対して「X」を ON
% ls -lg t1
-r-x--x--x  1 kuno  faculty  15 Mar 27 14:11 t1
%
```

このように、chmod コマンドを使うことで、さまざまな保護モードが自由に設定できます (なお、ls -l の表示の一番先頭はなぜか常に「-」ですが、これについては後で説明します)

2.5.3 i番号とディレクトリ

i番号は既に述べたように i ノードの番号で、領域配置層ではこの番号でファイルを把握しています。そして実は、普段使っている文字列の名前は「つけたし」であり、変更することができますし、また 1 つのファイルに複数の名前をつけることもできます。

- mv ファイル名 新しい名前 — 名前を変更する
- ln ファイル名 新しい名前 — ファイルに新しい名前をつける
- rm ファイル名 — 名前を無効にする

実際に行ってみましょう。

```
% ls
Library MH      Mail    t2
% ls -i                                       ↓ i-番号
168999 Library  223924 MH       202839 Mail     135188 t2
% cat t2
How are you?
% mv t2 t3  ←名前をつけかえても
% ls -i                                       ↓前と同じ
168999 Library  223924 MH       202839 Mail     135188 t3
% cat t3
How are you?
% ln t3 zzz  ←新しい名前をつけても
% ls -i                                        前と同じ↓
168999 Library  223924 MH   202839 Mail  135188 t3     135188 zzz
% cat zzz
How are you?
```

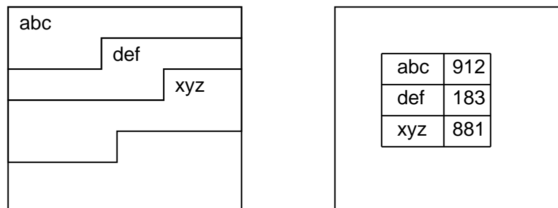

図 11: ファイルの名前はどこに入れればいいか?

```
% ls -l t3 zzz
-rw-------  2 someone        13 Apr 22 14:19 t3
-rw-------  2 someone        13 Apr 22 14:19 zzz
% rm t3      ↑この「2」というのは?
% ls -l zzz
-rw-------  1 someone        13 Apr 22 14:19 zzz
%            ↑持っている名前の個数
```

実は、rm はファイルではなく 名前を 消すコマンドです。そしてすべての名前が無効になったファイルは、アクセスできないので結果として消えます。すべての名前が無効かどうか知るには、「いくつ名前があるか」を記録すればよいわけで、この数も上のように ls -l の表示に含まれているのでした。

なぜこうなっているのでしょうか? そもそも、ファイル名はファイルと一緒に記録されているのでは ありません。もしファイル名がファイルと一緒に記録されていると、「このファイルを取りたい」と名前で指定したとき、ディスクの先頭から順に「このファイルかな? いや違った、ではこのファイルかな?」と捜して行かなければなりませんね? それではものすごく時間が掛かって役に立たないでしょう (図 11 左)。このため、名前はファイルとは別の場所に まとめて 表の形で記憶してあり、その表をさっと捜すとファイルの i 番号が分かるようになっているのです (図 11 右)。[13]この表のことを**ディレクトリ** (directory) と呼びます。[14]このように、コンピュータ内部の「しくみ」は全て「なぜそういうしているか」という理由があります。あなたがそのような「しくみ」の理由を分かっていないと、「動きはするけれども、ものすごくのろい」システムを作ってしまうも知れないのです。

2.6 ディレクトリ階層とパス名

ディレクトリからすぐにファイルにアクセスできることは分かりましたが、では (1) ディレクトリはどこに格納されていて、(2) どうやってアクセスするのでしょうか? (1) については、ディレクトリも構造は普通のファイルと同じであり、ただしその中身はファイルシステムが管理していて決まった内容 (つまりファイル名と i-番号の並び) が決まった形で入るだけになっています (ちなみに、「ls -l」の表示で一番最初が「d」になっているものはディレクトリです。あと 1 つ制約があり、ディレクトリは 1 つしか名前が持てません)。では (2) については? それは、まず i 番号の「2」は一番大元の**ルート** (root) ディレクトリであり、これは番号が決まっているので必ず取れます。[15]

ルートディレクトリには bin、usr などの標準的なものを置くディレクトリの i 番号が登録されています。bin は標準的なコマンド (ls とか ps とか) のプログラムを格納したファイルが入っていて、以下同様となっています。つまり、すべてのファイルはルートから始まるディレクトリの木構造 (階層構造) に組み込まれているのです (図 12)。[16]そして、ファイルを指定するのには**パス名** (pathname) が使えます。パス名は、「/」から始まる**絶対パス名** (absolute pathname) と、それ以外から始まる**相対パス名** (relative pathname) に分けられます。絶対パス名は、ルートから始めて名前の各成分 (/で

[13]i 番号からは簡単にデータブロックがアクセスできるのでしたね。
[14]Windows や Mac OS X では「フォルダ」と呼ぶことが多いです。
[15]では「1」は何かということになりますが、これは空いているブロックを管理するための i ノードになっています。
[16]実際には複数のディスクがあったりネットワーク経由でアクセスする場所があったりしますが、それは複数の木構造を「継ぎ合わせて」1 つの木構造にしています。

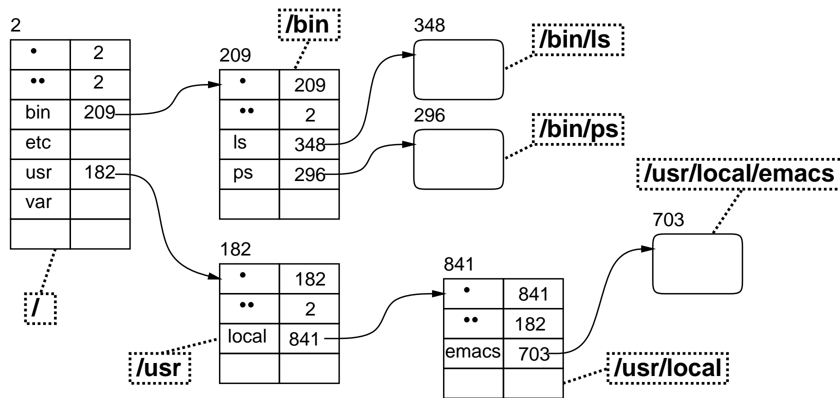

図 12: ディレクトリの木構造

区切られた個々の部分) を順番にディレクトリで探すことで目的のファイルやディレクトリに到達します。相対パス名の場合は、各プロセスごとに持っている「現在位置」ないし**カレントディレクトリ** (か) ら始めて、同様にたどって行きます。現在位置はコマンド `cd` で変更できます。

- `cd` パス名 — 指定したディレクトリを現在位置に
- `cd` — ユーザのディレクトリを現在位置に

また、図 12 を見ると、すべてのディレクトリに「`.`」「`..`」という項目が含まれ、「`.`」は「そのディレクトリ」、「`..`」は「1 つ上のディレクトリ」の番号に対応しています。これらにより、「このディレクトリ」とか「1 つ上に行ってそれからどちらへたどる」などの指定が可能になっているのです。

演習 2-4 自分の既に持っているファイルを `ls -l` で調べ、保護モードを確認しなさい。その後、以下の課題から 1 つ以上 (できれば全部) やってみてください。

a. 自分のファイルのモードを変更し、「読めない」ものの内容を (`cat` などで) 表示させようとしたり、「書けない」ものに (`cp` で内容をコピーするなどで) 書こうとした場合にどうなるか観察しなさい。できれば、「誰にでも読めるが自分には読めない」などの矛盾した設定の効果も調べるとなおよいです。

b. 他のクラスメートと協力して、他人のファイルについて、a. と同様に観察しなさい。できれば、グループや他人に対する保護設定のうまい活用例を考案し、実際にそれがうまくいくことを確認できるとなおよいです。[17]

c. 実行形式ファイルの「実行」について、その保護モードを OFF にするとどうなるか試しなさい。また再び ON にした場合どうなるか、実行形式でないファイルについて同様に ON にするとどうなるかも試しなさい。できれば、他のクラスメートと協力して、「その人には読めない (コピーできない) けれど実行できる」モードを実験してみて、それがどういう場合に役立つかを検討しなさい。

演習 2-5 ディレクトリにもファイルと同様に保護モードがあり、`chmod` コマンドで設定できます。たとえば、次のようにしてディレクトリを作り、その下にファイルを作ることができます。

```
% mkdir sub   ←ディレクトリを作る
% echo 'main() { puts("hello1."); }' >sub/test1.c    ←その下にファイルを作る
```

これについて、以下の課題から 1 つ以上 (できれば全部) やってみなさい。[18]

[17] 他人のホームディレクトリにあるファイルは我々の環境だと「`/u2/ユーザ名/ファイル名`」で指定できます。
[18] ディレクトリの保護モードを見るには「`ls -ld ディレクトリ`」とする必要があることに注意。

a. ディレクトリにも読み/書きの保護モードが設定できますが、これらはどのような効果を持つかについて実際にディレクトリ内にファイルを作って調べてみなさい。できれば、ファイルの読み/書きとディレクトリの読み/書きの違いについて検討できるとなおよいです。
b. クラスメートと協力して、他人のディレクトリにファイルを作ったりそこにあるファイルを調べるためにはどのように保護設定すればよいかを検討しなさい。できれば、複数の状況に対応した設定方法とその使い分け方についても検討できるとなおよいです。
c. ディレクトリの「実行」モードについては、ファイルの「実行」モードとは意味が違っています。どのように違うかを探究しなさい。できれば、読み/書きモードと「実行」モードの使い分け方についても検討できるとなおよいです。

「◎」の条件: (1) 小問を 2 問以上解答しており、(2) いずれも「できれば」の部分までやってあって、(3) 検討が筋道の通った内容になっていること。

演習 2-6 サーバ sma の上で、ルートディレクトリ「/」以下にどのようなディレクトリやファイルがあるかについて調べていき、結果を整理して示しなさい (一般ユーザでは読めない場所はそのように記してあればよいです)。できれば、自分の手元のマシンでも同様におこない、どのような違いがあるかも示せるとなおよい。

「◎」の条件: (1)「できれば」の部分までやってあって、(2) 調査が系統的であり (説明もきちんと書かれていること)、(3) まともな (と担当者の考える) 説明・考察が書いてあること。

3 まとめ

この回は、OS の概念からはじめて、プロセス、ファイルシステム、ファイル、保護モードパス名などを題材に OS の機能や考え方を学びました。また後半では、シェルやフィルタを題材に、Unix のツールがどのような考え方でできているかを見て頂きました。今回の内容は Unix が題材になっていますが、その考え方は他の OS やさまざまな場面で通用するものと考えてよいでしょう。