計算機科学基礎'11 ＃3 – ファイルシステムとデータ記憶

久野　靖*

2011.4.27

1　ストレージとファイルシステム

1.1　磁気ディスク装置とその性質

メモリは主記憶装置ともいわれるように、計算機システムにおける「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、今日の計算機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**ストレージ** (外部記憶装置) が使われます。ストレージ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、今日の計算機システムで広く使われているストレージ装置としては**磁気ディスク装置** (HDD、hard disk drive) 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**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 でもこれをディスクの替わりにする場合もあります (SDD、solid state disk) と呼びます。[1][2]

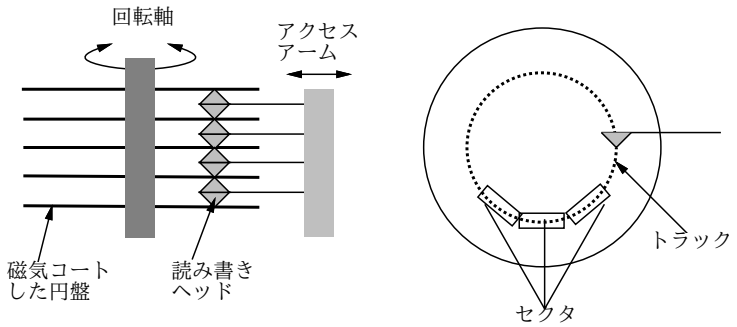

図 1: 磁気ディスク装置の構造

磁気ディスク装置の原理は図 1 のようなもので、回転する円盤 (**プラッタ**) の表面に (カセットテープ同様の) 磁気コーティングがしてあり、その上にヘッドを近づけて磁気的に情報を記録します。記

*経営システム科学専攻

[1]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。

[2]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、コンパクトフラッシュなどさまざま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)。

録はらせん状ではなく同心円状におこない、その 1 つの同心円を**トラック**と呼びます。ヘッドと一体になったアクセスアームを半径方向に移動し、複数あるヘッドのどれかを電気的に選択することで、任意のトラックを比較的高速に選択できます。また、トラック内のデータブロックを**セクタ**と呼びます (1 セクタは通常 512 バイト)。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。[3]

たとえば、2TB で 1 万円のドライブを考えると、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 (4GB で千円くらい?) 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。なぜかというと、それはディスクが回転していることと関係があります。毎分 7200 回転 (7200RPM —- revolutions per minutes) のディスクの場合、1 回転するのに $60sec/7200 = 8.33msec$ を要します。どんなに頑張っても、読もうと思うデータが今ヘッドの下を行き過ぎたばかりだと、1 周して再びヘッドの下まで回ってくるのに 8msec 待つ必要があるわけです。実際には「幸運な」場合もあるわけで、平均回転待ち時間は半分の 4msec ですが、これにヘッドを読み書きするトラックの位置まで動かす時間 (これも「動かす遠さ」に依存しますが、平均 8msec とかでしょうか) も加える必要があります。

これらを合わせると、主記憶装置の待ち時間が 80nsec 程度なのと比較すると、ディスクアクセスの待ち時間はその 10 万倍もの時間、ということになります。このことは、ストレージの利用のしかたに対して大きな影響を与えます (また後で取り上げます)。

1.2 ファイルとファイルシステム

ここまでで学んだように、磁気ディスク装置に対してできることは「どのトラックのどのセクタを読め/書け」と指示することだけです。しかし、そのような裸のディスク装置を「そのまま」使えと言われたら非常に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 (大きな紙を壁に貼って各自の割り当てを書き込む?)。
- 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。

[3]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。

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。

これはちょうど前回学んだ、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だと考えればいいでしょう (ファイルの数の方がプロセスよりはるかに多いですが)。つまり、ディスク装置など裸のストレージの上にコンピュータによって作り出された、**仮想化**された使いやすいストレージ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**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、次の 2 つの部分から成っています (図 2)。

- **領域配置** (storage layout) 層 — 記憶領域の割り当てや配置を管理する。
- **属性管理** (attribute handler) 層 — 名前や保護設定などファイルの各種情報を管理する。

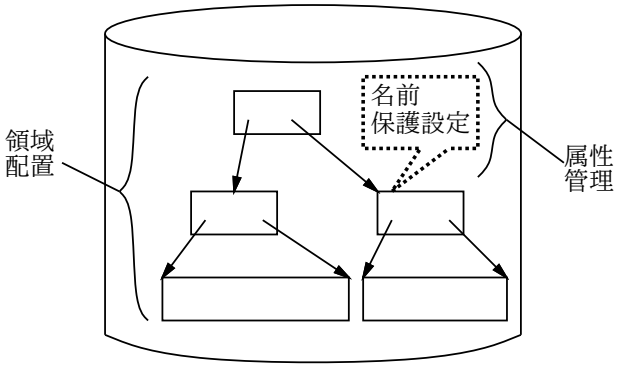

図 2: 領域配置と属性管理

1.3 属性管理と領域管理の区分 option

一般に、領域配置層は媒体に応じて変わります。たとえば、磁気ディスクと DVD とフラッシュメモリでは、読み書きの特性がまったく違いますから、領域の割り当てや配置はそれぞれに合わせる必要があるのです。これに対し、属性管理は「どのように名前をつけるか」「どのように保護設定を管理するか」ですから、媒体が違っても共通です。このため、OS の内部では領域配置層が「下側」属性管理層が「上側」になります。

また、(たとえば Unix と Windows のように)OS が違うと、名前の付け方や保護機能の内容が変わりますから、属性管理層が違って来ます。しかし、媒体が同じであれば領域配置層の機能はだいたい共通しています (たとえば CD-R や DVD などでは配置方法に完全に互換性があります。さもないと Unix で書いた媒体が Windows で読めないということになりますから)。

以下ではまず領域配置層、続いて属性管理層についてその概要や重要な機能について見て行くことにします。

ところでその前に 1 つだけ…ファイルの「名前」が属性管理層の役割だとすると、下側 (土台側) の領域配置層で扱うファイルには名前がないのでしょうか? これは YES であり NO です。つまり、人間が扱うような、文字列の名前はありません。しかし全く名前がないと不便ですから、領域配置層では名前は「番号」になっています。番号つまり整数は計算機上では効率よく扱えるので、OS の基幹部分にある領域配置層ではその方が好都合なのです。たとえば Unix の場合は、後で出て来る **i 番号**と呼ばれる値が領域層でファイルを識別する「固有番号」になっています。このような、OS 内部では番号、人間が扱うのは文字列という使い分けは、ネットワークなどでも見られます。

2　領域配置層とその機能 option

2.1　ユーザデータ/メタデータ/スーパブロック option

領域配置層の役割りは、ファイルの領域とファイルでない (つまりこれ空いている) 領域を把握し、適切に割り当て/解放を行うことです。また、「何番のファイルの何バイト目」と言われた時にそれはディスクのどのトラックのどのセクタかを計算できることも必要です (さもないとデータの読み書きができませんから)。

そして、電源断があるとメモリ上の情報は失われてしまうので、常にディスク上の情報だけで確実にこれらの情報が取り出せるようにしておくことも大変重要です。突然の停電から復帰してみたらファイルがまったく無くなっていた、ではお話になりませんから…

ディスク上にあるファイルシステムのデータは、次の 2 種類に分かれています (図 3)。

- **ユーザデータ** — 格納すべきファイルの「中身」「内容」。
- **メタデータ** — 管理情報。ユーザデータ以外のすべての情報。

つまり、どこに何番のファイルがあるか、どこが空いているか、などの情報はすべてメタデータということになります (このほか、属性管理層が扱う情報もすべてメタデータということになります)。そして、それらのメタデータ自体もディスクの「どこかに」格納するわけですから、それらがどこにあるかの情報も必要です。

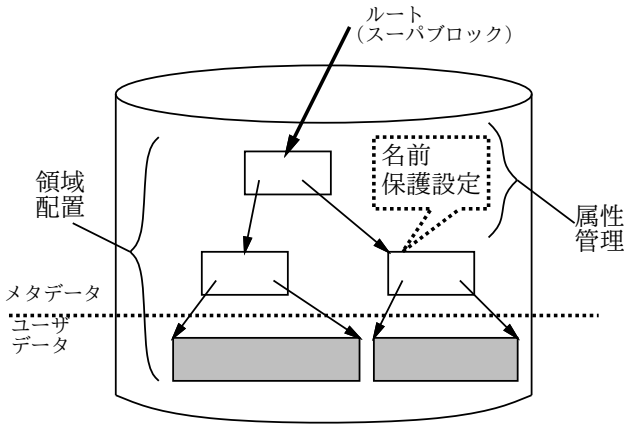

図 3: ユーザデータ/メタデータ/スーパブロック

そうやって遡って行くと、すべての情報の「根元」にあたる情報がどこかにあることになります。これを**スーパブロック**と言い、「ディスクの一番先頭」「ディスクのサイズをもとにしてある決まった計算方法で計算した場所」など、計算だけで求まる場所に置かれています。つまり、ディスク上の情報はすべて、スーパブロックから順次たどって行ける枝分かれした**木構造**として格納されているわけです。

2.2　Unix のファイルシステム option

ではもう少し具体的に、Unix のファイルシステムを例に領域配置層のデータ構造を見て行きましょう。なお、Unix のファイルシステムは (特に Linux 系を中心に) さまざまな新しいものが開発されて来ていますが、ここでまず説明するのはもっとも古くからある原理部分です (それが一番分かりやすいので)。その後で、新しいファイルシステムではどのようなことが改良されているかについてもあらすじだけ説明します。

Unix では伝統的に、1 つのファイルごとに対応するメタデータを **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**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。[4]

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。スーパブロックはディスクの一番先頭にあり、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 4)。

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 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。

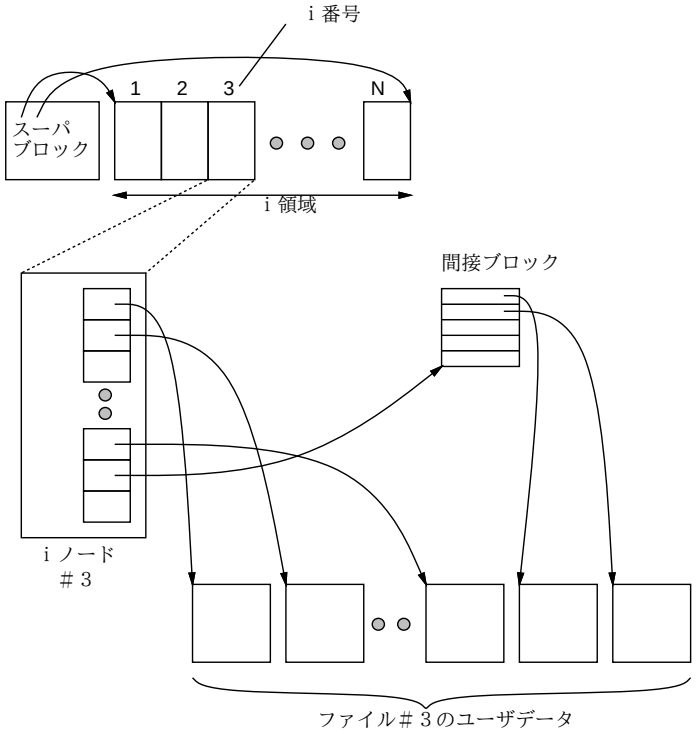

図 4: i ノードとデータブロック

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。[5]

さて、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、それではディスクの性能がろくに発揮できない (本来の読み書き性能の 5%とか) ことが分かって来ました。なぜかというと:

- i 領域がディスクの先頭にあるのに対し、データブロックはディスクの中央以降にある。ファイルのアクセスには i ノードとデータブロック両方のアクセスが必要なので、必ず途中でヘッド

[4] ブロックサイズはファイルシステムを初期化するときに指定可能で。大きくした方が読み書きがまとめてできるので高速になるが、その代わり小さいファイルを沢山作ると「ブロックの使っていない部分」のための無駄が大きくなる。

[5] OS によっては…たとえば FreeBSD では、三重間接ブロックまで用意してさらに大きなファイルに対応している。

を動かすはめになり待ち時間が長くなる。

- しかも i 領域が「はしっこ」にあるので常にヘッドが「はしっこ」に来てしまう。
- あきブロックはバラバラに割り当てられるので、1 ブロック読むごとにやはりヘッドを動かす必要がある。
- i ノードを新しく割り当てる時は先頭から順に「空いている場所」を順次探して行く必要があり、手間がかかる。

この問題を解決するため、FFS(Fast File System) というのが考案されました。そこでは、データ構造はこれまでと同様ですが、以下のような改良が施されました。

- 「シリンダグループ」と呼ばれるトラックの集まり (ヘッドをあまり動かさなくて済む範囲) を作り、それぞれのシリンダグループに i 領域とデータブロックを配置する。そして、ある i ノードのデータブロックは同じシリンダグループから割り当てることで、ヘッドの動きを少なくした。
- あきブロックを割り当てるときになるべく連続して割り当てることで、連続して (ディスクの最高速度で) 読み書きができるようにした。
- あき i ノードがすぐ分かるように、i ノードの空き/使用状況を各々の i ノードについて 1 ビットで表した「ビットマップ」を用い、これを見れば空いているものがすぐ分かるようにした。

現在の FreeBSD のファイルシステムも基本的な構造はこれを継承していますが、ただし電源断などの際の整合性管理について改良がなされてきています。そのほかの OS(Linux など) でも現在、さまざまな機能を盛り込んだファイルシステムが研究され実用化されてきています。

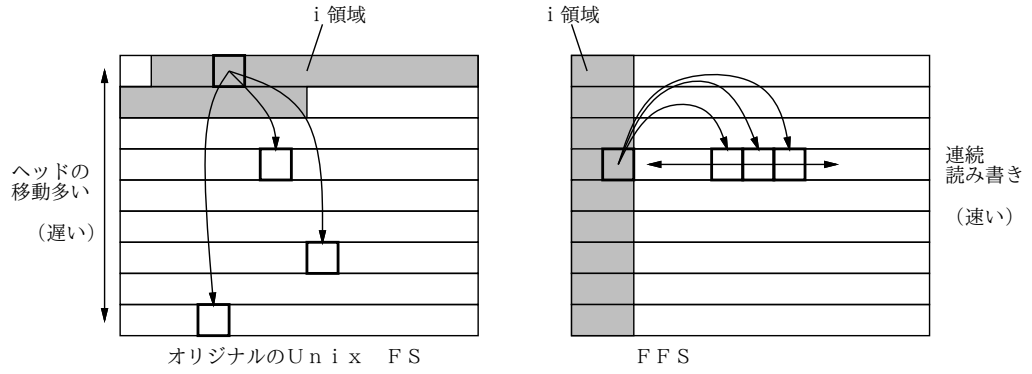

図 5: Fast File System

3　属性管理層とその機能

3.1　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
Mail     WWW      t         t.c       t.s
% ls -a
.              .canna32       .mh_profile    .trash          WWW
..             .emacs         .mime.types    .twmrc          t
.bash_history  .fullcircle    .mnews_setup   .x11defaults    t.c
.bash_logout   .gtkrc         .mozilla       .xfce           t.s
.bash_profile  .login         .netscape      .xfm
.bashrc        .logout        .newsrc        .xinitrc
.blackboxrc    .mailcap       .profile       Mail
%
```

「.」で始まるファイルが多いのに驚きましたか? 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ます。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ようになっているのです。

3.2 長さ

ファイルには、長さがあります (当然ですね)。OS によっては、ファイルの長さが 512 バイトなどの「かたまり」単位でしか決まらないものもありますが、Unix ではファイルの長さは 1 バイト単位で決まっています。長さを調べるには、`ls` を次のように使います:

- `ls -l` — 長さを始めとする詳しい情報を表示。

ところで、次のような C プログラムを格納したファイル `t.c` があるものとしましょう。

```
% cat t.c
main() {                    ←  8 文字
  puts("Hello.");           ← 17 文字
}                           ←  1 文字
%
```

そして C プログラムに現れる各文字は、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 t.c
-rw-r--r--   1 someone        29 Mar  1 13:06 t.c
%
```

答えは「29 バイト」ですが、合っていましたか? これを見ると、まず、「空白」も文字として数えることが分かります。しかし空白を入れてもまだ目に見える「文字」の総数よりファイルの長さが 3 バイト多いことが分かりますが、これはそれぞれの行の終りに**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。

3.3 日付

`ls -l` ではそのファイルを最後に変更した日時も表示されるので、そのような情報がファイルに付属していることは分かります。実はそれ以外に、最後に読み出した日時も記録されています。

- `ls -t` — 表示を変更日時の新しい順におこなう。
- `ls -lu` — `-l` と同じだが変更日時でなく読み出し日時を表示

なお、ls コマンドを使う時には、これらを自由に組み合わせることができます。たとえば-a と-l と -t をまとめて指定する時は「ls -l -a -t」でもいいのですが、もっと短くして「ls -lat」で構いません。さらに、既に何回も使っているように、ファイル名を 1 個以上指定すればそのファイルに関する情報のみが表示されます。たとえば次のようになります。

```
% ls -l
total 12                                  通常は ABC 順↓
drwx------   4 someone       512 Mar  1 12:48 Mail
drwx--x--x   2 someone       512 May 29  2001 WWW
-rwx------   1 someone      6484 Mar  1 13:11 a.out
-rw-r--r--   1 someone         0 Mar  1 13:11 t
-rw-------   1 someone        29 Mar  1 13:06 t.c
-rw-------   1 someone       383 Mar  1 13:06 t.s
% ls -lut
total 12                        読み出し時刻順になった↓
-rw-------   1 someone       383 Mar  1 13:11 t.s
-rwx------   1 someone      6484 Mar  1 13:11 a.out
-rw-r--r--   1 someone         0 Mar  1 13:08 t
-rw-------   1 someone        29 Mar  1 13:07 t.c
drwx------   4 someone       512 Mar  1 12:48 Mail
drwx--x--x   2 someone       512 Mar  1 01:07 WWW
%
```

3.4 持ち主、所属グループ

ls -l ではそのファイルの持ち主 (作った人) も表示されます。なぜ、持ち主の情報が必要なのでしょう? それはもちろん、「持ち主には読めるが他の人には読めない」といった保護を行うには、持ち主が記録されていないと困るからです。さらに Unix では、保護を柔軟に行うために「持ち主」と「その他の人」の中間として「グループ」というものが定義されています。そして、ファイルには持ち主に加えて、所属グループの情報も付属しています。これを見るには次のものを使います:

- ls -lg — -l と同じだが、所属グループも表示

いっぽう、各ユーザは 1 つ以上のグループに所属しています。[6]自分が所属しているグループはコマンド **groups** で表示させられます:

- groups — 自分が所属しているグループの一覧を表示

そして、ファイルの持ち主は **chgrp** でファイルのグループを変更できます:

- chgrp グループ名 ファイル … — ファイルのグループを変更

指定できるグループは自分が所属しているグループのどれかに限られます。

```
% groups                ↓私 (久野) は 4 つのグループに所属
faculty wheel operator staff
% ls -lg t1             ↓普段作るファイルは教官グループ
-rw-r--r--  1 kuno     faculty       894 Apr 22 14:03 t1
% chgrp wheel t1
% ls -lg t1             ↓だが管理者グループに変更もできる
-rw-r--r--  1 kuno     wheel         894 Apr 22 14:03 t1
%
```

[6]古い Unix ではちょうど 1 つだけのグループに所属していましたが、それだと不便なので現在では複数グループに所属できます。

3.5 ファイルのモード

Unix では、ファイルの保護設定をモードと呼び、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)

かどうかをそれぞれ設定できます。モードの情報は `ls -l` の表示の最初の部分に含まれています。実は先に出てきた

```
-rwx------  1 someone      7456 Apr 22 13:58 a.out
```

というのは、持ち主 (someone) は読み、書き、実行ともに可能だがそれ以外の人にはどれも不可能という設定を意味しています。これに対し

```
-rw-r--r--  1 kuno     faculty       894 Apr 22 14:03 t1
```

というのは、持ち主 (kuno) は読み書きともに可能ですが、グループ faculty の人、およびその他の人には読むことだけ可能という設定を意味しています。

モードを変更するのには、**chmod** コマンドを使って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 のどれか 1 つ以上 (それぞれ上記の持ち主、グループメンバ、その他の人に対応)、「許可」は r、w、x のどれか 1 つ以上 (それぞれ上記の読み、書き、実行に対応) で、「+」を指定するとその許可を出すこと、「-」を指定するとその許可を取り除くことを意味します。たとえばさっきのファイルでやってみましょう。

```
% ls -lg t1
-rw-r--r--  1 kuno     faculty       289 Apr 22 14:07 t1
% chmod ugo-rwx t1
% ls -lg t1
----------  1 kuno     faculty       289 Apr 22 14:07 t1
% chmod u+rx t1
% ls -lg t1
-r-x------  1 kuno     faculty       289 Apr 22 14:07 t1
% chmod go+x t1
% ls -lg t1
-r-x--x--x  1 kuno     faculty       289 Apr 22 14:07 t1
%
```

なお、`ls -l` の表示の一番先頭はなぜかいつも「-」になっていますね。これについては後で説明します。

3.6 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?
% 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**はファイルを消すコマンドではなく、名前を消すコマンドだったのです。そして、すべての名前が無効になったファイルは、それ以上触りようがなくなるので結果として消えます。すべての名前が無効になったかどうか知るには、ファイル毎に現在いくつ名前を持っているか記録しておけばよいわけです。この数も上の例のように、`ls -l`の表示に含まれているのでした。

なぜこういうふうに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したら、「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?」と読みながら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6左)。

このため、名前はファイルとは別の場所に**まとめて**表のような形で記憶してあり、その表をさっと捜すとファイルのi番号が分かる、という風になっているのです (図6右)。そして、この表のことを「ディレクトリ」と呼びます。i番号から効率よくデータブロックをアクセスする仕組みについては既に説明しました。

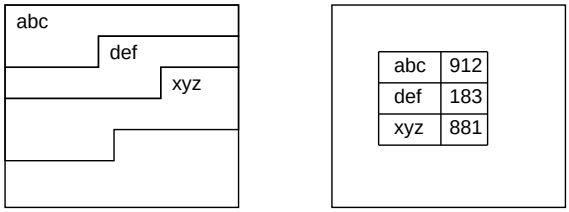

図 6: ファイルの名前はどこに入れればいいか?

このように、計算機の内部の「しくみ」はそれぞれ「なぜそういう風にしているか」という理由があります。逆にあなたが (自分がプログラミングするにせよ、誰かに発注するにせよ) そのような「しくみ」の勘どころを分かっていないと、動くことは動くけれども、ものすごくのろい、というシステムを作ってしまうかもしれないわけです。

4 入出力とシステムコール

4.1 入出力のプログラミング

ここまででではもっぱら、コマンドで操作している時ファイルがどう見えるかについて説明してきました。本節では、プログラムの中からはファイルがどのようにして操作されるかを見てみましょう。

まず重要なことは、ファイルの読み書きを各プログラムが直接ディスク装置への指令という形で行うことはできないということです。これは、もしそんなことを許すと、誰もが間違って他人のデータを読んでしまったり書き換えてしまえます。ではどうするかというと、OS に「これこれのファイルを読みたい/書きたい」と頼み、あとは OS 内部のファイルシステムがその可否を (先に出てきた保護モードなども含めて) チェックして OK な場合だけ頼まれた操作を行うのです。この「OS に作業を頼む」ことをシステムコールと呼びます。

Unix で C 言語を使っているぶんには、システムコールは C のサブルーチンであるかのように記述できます。入出力のための基本的なシステムコールは次の 2 つです。

```
read(ファイルディスクリプタ, バッファ, バイト数)
write(ファイルディスクリプタ, バッファ, バイト数)
```

ここで**ファイルディスクリプタ**というのは入出力の対象を表すための小さな整数であり、標準では 0 番、1 番、2 番というの 3 つのディスクリプタが用意されます。`read`/`write` とも戻り値として、実際に読み書きしたバイト数を返します。`read` は、ファイルの終りまで来るとそれ以上読めなくなるのでバイト数として 0 を返します。`write` ではエラーがない限り、引数で指定したバイト数と同じ値が返ります。

3 つのディスクリプタはそれぞれ次のように割り当てられています。

0 : 標準入力。特に指定しなければキーボードを意味する。
1 : 標準出力。特に指定しなければ端末画面を意味する。
2 : 標準エラー出力。特に指定しなければ端末画面を意味する。

ディスクリプタというのは簡単にいえば、プロセスが外部とデータをやりとりするための「通路 (チャネル) の番号」です (図 7 左)。そして、特に指定しなければこのチャネルは画面やキーボードに接続されているわけです (図 7 右)。

4.2 リダイレクトと機器独立性

ではさっそく、簡単なプログラムを見てみましょう。次は、簡単なメッセージを画面に打ち出すプログラムです。

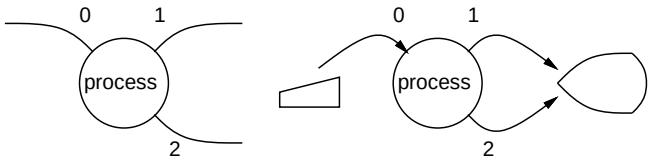

図 7: ファイルディスクリプタと入出力

```
/* write.c -- simple use of write */
main() {
  write(1, "This is a pen.\n", 15);
}
```

ところで、前回やったプログラムでは、メッセージを打ち出すのに printf というのを使っていました。この違いについては後で説明します。ともあれ、これを動かしてみましょう。

```
% gcc write.c
% a.out
This is a pen.
%
```

特に問題ありませんね? さて、実はこの「チャネル」の接続先を変更するように指定できます。

```
% a.out >t1
%
```

コマンド行に「>ファイル名」のように指定すると、標準出力 (チャネル 1 番) の接続先は指定されたファイルに切り替わります。だから画面には何も現われませんが、代わりに出力はファイルに書き込まれています。

```
% cat t1
This is a pen.
%
```

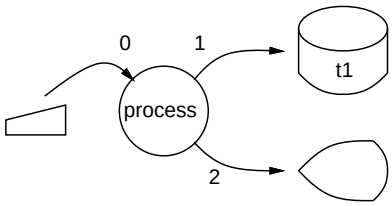

図 8: 出力の切り替え

これを Unix では出力の**リダイレクション**と呼びます。この様子を図 8 に示します。ところで 2 番 (標準エラー出力) は何のためにあるのでしょうか? それは、出力をリダイレクトしてもエラーメッセージなどは画面に見えて欲しいからです。図 8 にあるように、1 番をリダイレクトしても 2 番は依然として画面につながっているので、エラーメッセージを 2 番に書けばそれは画面に現われるのです。

ところで、ここでもう 1 つ重要なのは、プログラム自体は何ら変更しないままで、その出力先を画面にしてもファイルにしても動かすことができた、ということです。つまり、相手が「出力できる先」でさえあれば、それが具体的に何であってもプログラムは同じままでよいのです。このような性質を (入出力の) **機器独立性**といいます。このような性質のおかげで、画面に出力する時とファイル

に書く時とネットワーク経由で転送する時とで別々のプログラムをいちいち用意しなくてもよくなるわけです。

もう 1 つリダイレクションの例を挙げましょう。

```
% a.out >/dev/tty
This is a pen.
%
```

こんどはリダイレクトしたはずなのに画面に出て来ましたね。これは、/dev/tty というのが**端末デバイス**つまり画面+キーボードにつながっている「特別なファイル」だからです。実は cp コマンドでコピー先を/dev/tty にしてもやはり出力が画面に現われます。逆に、cp コマンドを使ってキーボードから打ち込んだものをファイルに取りこむこともできます。このように、ファイルと同様に扱えるけれども実際には入出力機器につながっているものを、Unix では**デバイスファイル**と呼びます。

```
% echo ABC >t1
% cat t1
ABC                ←ファイル t1 の中身は「ABC」
% cp t1 /dev/tty   ←/dev/tty にコピーすると…
ABC
% cp /dev/tty t2   ←/dev/tty からコピー…
XYZ
^D                 ← Ctrl-D は「ファイル終わり」の意味
% cat t2
XYZ                ←確かに入力できた
%
```

この機能があると、ファイルへの入出力と機器への入出力とが同じプログラムで行えますから、機器独立性がより増すことになります。

4.3 入出力のスループット

さて、15 バイトくらいの出力ならどういう風にやっても問題はありませんが、もっと大量のデータを扱う時には色々考えなければならないことが出てきます。次のプログラムは、ループを使って 100MB の出力を行ないます。

```
/* largefile.c --- output 100MB */
char buf[1024];
main() {
  int i = 0;
  while(++i <= 100000)
    write(1, buf, 1024);
}
```

100MB のデータをディスクに書き出すのにどれくらい掛かると思いますか? 言い替えれば、現在の普通のディスクのスループット (時間あたり実効転送量) はどれくらい (何 MB/s) だと思いますか?

```
% time a.out >/var/tmp/t
real    0m2.240s
user    0m0.019s
sys     0m0.624s
%
```

約 2.2 秒、ということはスループット 45MB/s くらいということですね。ところでなぜ/var/tmp/t というところに出力したのか分かりますか?

```
% time a.out >t
real    0m3.777s    ← 26MB/s???
user    0m0.049s
sys     0m1.041s
% time a.out >/tmp/t
real    0m0.793s    ← 125MB/s???
user    0m0.028s
sys     0m0.615s
% time a.out >/dev/null
real    0m0.089s    ← 1111MB/s???
user    0m0.014s
sys     0m0.074s
%
```

/var/tmp というディレクトリ (ファイルの置き場所) に作業ファイルを作る場合は、そのマシンのローカルディスクが使われるので、ディスクのスループットが計測できます。

しかし、ディレクトリによってはローカルディスクではないところに書いてしまうこともあります。たとえば、ホームディレクトリはサーバ上にあってネットワーク経由でアクセスしているという環境だと、ホームディレクトリ上にファイルを作るとスループットはおもにネットワークの転送速度で決ってしまいます。筆者のサイトでは転送速度が 1Gbit/s(ということは 100MB/s くらい) のイーサネットが主に使われていますが、実際にこれを通してファイル等を転送しようと思うとスループットは 25MB/s 程度になってしまいます。

また、/tmp/t というのは何でしょう? これは「メモリファイルシステム」つまり作業用ファイルをディスクではなくメモリに取るというものなので、ディスクへの入出力は一切起こりません。ディスクとメモリでは 1 桁くらいスループットが違うのでこの結果になるわけです。では最後の**/dev/null** というのは何でしょう? これは「書き込まれたデータを黙って受け取って捨ててしまう」仮想デバイスなので、データの転送すら起こりません。つまりデータの転送を除いたプログラムの実行時間が計れてしまうわけです。

ここから何が分かるでしょうか? 現在の計算機システムでは、先に述べた機器独立性や**分散透明性** (ネットワークの向うにあるものでも手元にあるのと区別なく使える性質) のおかげで、使うだけなら「どこの何を使っているか」は意識しなくても動作し、大変便利です。しかしこと性能となると、やはりネットワーク経由は遅いし、メモリだけで済めば圧倒的に速いわけです。このことを分かっていて意識して使わないと、「動くけどめちゃくちゃのろい」結果になってしまうわけです。

たとえば今作ったカレントディレクトリの 100MB のファイルを別のファイルにコピーする、という作業を考えてみます。まず手元のマシンでやってみました:

```
% time cp t t2
real    0m6.554s
user    0m0.002s
sys     0m0.478s
%
```

6.5 秒ということは、読み書き速度は合わせて 15MB/秒? ずいぶん遅いようです。そこで今度はファイルサーバ上でやってみました:

```
% time cp t t2
real    0m1.123s
```

```
user    0m0.001s
sys     0m0.564s
%
```

90MB/秒とだいぶ速くなりました。つまり、自分のホームディレクトリはファイルサーバ上にあるので、手元のマシンでやるとネット経由で 100MB を読んで書いていたわけで、実はネットワークの速度を計っていたようなものですね。これに対し、ファイルサーバでやればそこにディスクがあるので、ディスクの速度が計れます。ところで、もっと速くする方法があるのは分かりますか?

```
% time mv t t2
real    0m0.079s
user    0m0.000s
sys     0m0.040s
%
```

mv はファイルの名前を付け変えるだけですから、データのコピーは一切起こりません。つまり、コピーしなくていいものはコピーしないに越したことはない、ということです。

5 ファイルシステムと名前空間

5.1 ディレクトリ再び

これまで、ログインしたらそこで ls コマンドにより自分のファイルが見えることを当たり前のように考えてきましたが、これはよく考えてみると不思議ではないでしょうか? つまり A さんがログインして仕事をしている間は A さんのファイルが見え、B さんがログインすると今度は B さんのファイルが見えるわけですから。なぜ二人のファイルはまぜこぜになってしまわないのでしょう?

答えはある意味非常に簡単で、先に述べたディレクトリが個人ごとに別に用意されているから、というのが答えなわけです。そしてもう 1 つ、すべてのプロセスには**現在位置** (カレントワーキングディレクトリ) が対応していて、特に指定しない場合、新しくファイルを作ればそのディレクトリにできるし、ls もそのディレクトリにあるファイルの一覧を表示するわけです。この様子を図 9 に示します。

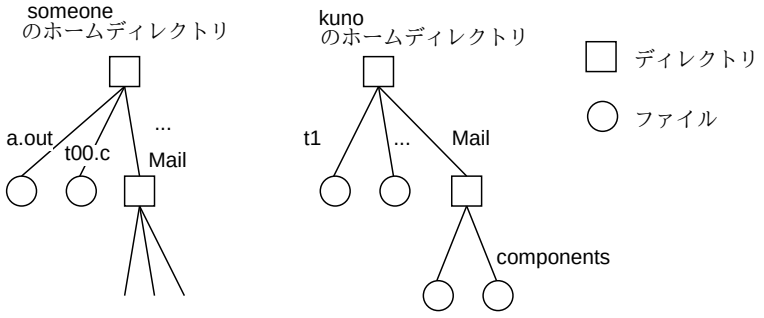

図 9: ディレクトリの概念

ではなぜ、A さんと B さんが使っている時それぞれのカレントディレクトリが違っているのでしょう? それは、ログイン処理を担当する部分で、ユーザ名に応じてそれぞれ固有のディレクトリを現在位置にしてからコマンドインタプリタを起動するからです。この、ログイン時の現在位置を各自の**ホームディレクトリ**と呼びます。

ところで、ディレクトリにはファイルだけでなく他のディレクトリを入れることもできます。ディレクトリの中に入っているディレクトリを**サブディレクトリ**呼びます。たとえば図 9 では、ユーザ kuno は Mail というサブディレクトリを持っていて、その下に 2 つファイルを持っています。サブディレ

クトリの下にさらにディレクトリを作って、階層をいくらでも深くすることができます。サブディレクトリを作る/消すには **mkdir** と **rmdir** を使います:

- `mkdir` ディレクトリ名 — ディレクトリを作る
- `rmdir` ディレクトリ名 — ディレクトリを消去する

ディレクトリは `ls -l` ではモード表示の最初に「d」と表示されるのでそれと分かりますし、また `ls -F` による表示では名前のあとに「/」がついて表示されます。

5.2 ディレクトリの木構造

ところで、Unix のファイルシステムは図 9 のようなホームディレクトリがふわふわ沢山浮かんでいるものだ、と思う人はあんまりいないでしょうね。実際には Unix のファイルシステムには一つだけ**ルートディレクトリ**と呼ばれるディレクトリがあり、これが文字通りディレクトリの木の「根っこ」になっています。すべてのディレクトリやファイルはルートディレクトリの「子孫」にあたります (図 10)。

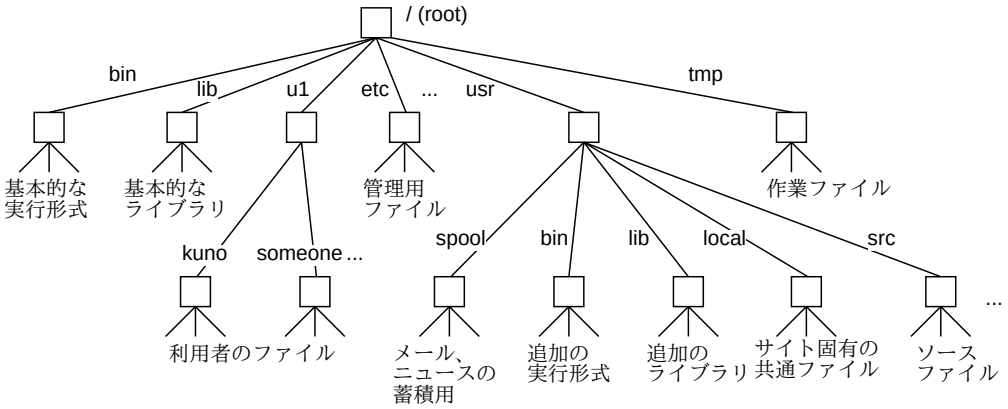

図 10: ディレクトリの木構造

図 10 の中には、`lib` とか `bin` という名前のディレクトリが複数ありますし、また各自が同じファイル名を考えつくこともあります。もちろん、これらはディレクトリの木の中で別の場所にある別のものですが、それらを区別して指定する方法が必要ですね。それには**パス名**というものを使います。パス名には次の 2 通りがあります。

```
/名前/名前/.../名前      -- 絶対パス名
名前/名前/.../名前       -- 相対パス名
```

絶対パス名というのは、ルートから始めて指定した名前を順番にたどることで目的のファイルやディレクトリの位置が示されることを意味しています。たとえば次のような感じです。

```
/u1/someone/a.out          : someone さんのホームディレクトリにある a.out
/u1/kuno/Mail/components : kuno さんの Mail サブディレクトリ下の components
/                          : ルートディレクトリそのもの
```

一方相対パス名というのは、ルートの代わりに現在位置から始めて同様にたどることを意味します。ですから、現在位置が someone さんのホームディレクトリであれば単に `a.out` で someone さんのホームディレクトリの下の `a.out` を意味します。つまり、これまで「ファイル名」と思っていたのは実は「パス名」の特別な場合だったのです。[7]

ところで、パス名の成分には特別な名前として次の 2 つが使えます:

[7]なお、少なくとも 1 つは「/」を含んでいないとパス名とは呼ばない、という流儀もあります。

```
.       : そのディレクトリ自身
..      : そのディレクトリの一つ上
```

たとえば同じく someone さんのホームディレクトリにいる場合だと次のような具合です:

```
.                          : 自分のホームディレクトリ
Mail/components            : 自分のホームにある Mail の下の components
../kuno/Mail/components : kuno さんの Mail サブディレクトリ下の components
../..                      : ルートディレクトリ
```

相対パス名は絶対パス名より短く指定できるので、ある場所にあるファイル等をたくさん操作する場合は、そこへ現在位置を移動してから作業するのが一般的です。そのための指令として、**pwd** と **cd** があります:

- `pwd` — 今いる所 (現在位置) の絶対パス名を表示する
- `cd` パス名 — 指定した場所に行く (つまり、現在位置にする)
- `cd` — ホームディレクトリに行く

`cd` でそのディレクトリへ「行けば」、`cd` コマンドを打つ手間が掛かる代わりに、そこにあるファイルは名前だけで指定できるわけです。

5.3 保護設定ふたたび

ディレクトリもファイルと同様に保護モードを持っています。ディレクトリの保護モードを調べたい場合は、次の形の `ls` を使ってください:

- `ls -ld` ディレクトリ — ディレクトリの情報を表示

オプション「`-d`」がないと、`ls` はディレクトリ自体の情報ではなく、その中に入っているファイルの一覧を表示してしまいます (それも確かに必要な機能ではあります)。では例を見ましょう:

```
% ls -ld work
drwxr-x--x  3 kuno  staff  1024 Jul  2  2002 work
%
```

これを見ると、ディクトリ `work` の保護モードは…まず先頭に「`d`」がありますが、これはディレクトリであることを表しています (ファイルでは「`-`」でしたね)。

次に、このディレクトリは所有者には読み、書き、実行ともに可能です。ディレクトリでは、「実行」とは「そのディレクトリをたどって中のファイルやディレクトリを参照できる (もちろん、そのファイルなりの保護モードが許せば、です)」という意味になります。ついでに確認しておくと、「読める」とは `ls` などで一覧を表示できる、という意味であり、「書ける」とはそこに新しいファイルを作ったり、既存のファイルを削除したりできる (当然そういうことをするとディレクトリの内容が変わりますから書き換える必要があるわけです)、という意味になります。ですから、ディレクトリ `work` は所有者には何でも操作でき、グループ `staff` のメンバには一覧を表示したり中のファイルをアクセスでき、それ以外の人には中のファイルをアクセスできる、ということになります。2 番目と 3 番目の違いが分からない、ですか? つまり、3 番目の場合、一覧が取れないのですから、ディレクトリに入っているファイルの名前を正確に知っている人だけがそのファイルを参照できる、という形の保護になるわけです。

今日では Web サーバによる情報発信が一般化しているので、それとファイル/ディレクトリ保護との関連についても説明しておきましょう。Web サーバは、ある特定のディレクトリ (**公開ディレクトリ**) 以下のファイル (通常は HTML ファイルですがテキストファイルでも何でも) を、ブラウザから

の要求に応じて送出します。ですから、公開ディレクトリ以下に機密のファイル (重要なデータや読まれるとまずい設定ファイル等) を置いてはいけません。Web サービスのための実行ファイルがこれらのファイルを必要とする場合は公開ディレクトリの「外に」置くべきです。

Web サーバはユーザ「nobody」などの専用ユーザとして実行されるので[8]、Web サーバが読むファイルはこのユーザに読める保護設定にしておく必要がありますが、書き込みはできなくしておくべきです。ただし、掲示板などデータを保存する Web アプリケーションではその専用ユーザがファイルに書き込めるようにする必要があるかも知れません。その場合も、その特定ファイルのみ書き込み可能とし、ディレクトリなどに自由に書けるようには設定するべきでありません。

このあたりの保護設定は外部に公開するサーバでは慎重な管理が必要ですので、実際にそのようなケースに遭遇したら参考書などを参照してきちんと計画してください。

5.4 名前空間とマウント option

このように Unix では、全てのファイル/ディレクトリ/入出力装置は一つの木の形に編成されていますが、実際には一つのディスクに全てのファイルを収めることはディスク装置の容量によっては難しいことがありますし、管理上も不便なことがあります。

そこで、Unix ではディスク (正確にはディスクをいくつかに区切って使う、その 1 **区画**) ごとにディレクトリの木が存在し、それを張り合わせる (**マウント**する、という) ことで一つの木に構成するようになっています。[9]この様子を図 11 に示します。ネットワーク共有もこの「貼り合わせ」のしくみを用いて行なえるわけです。

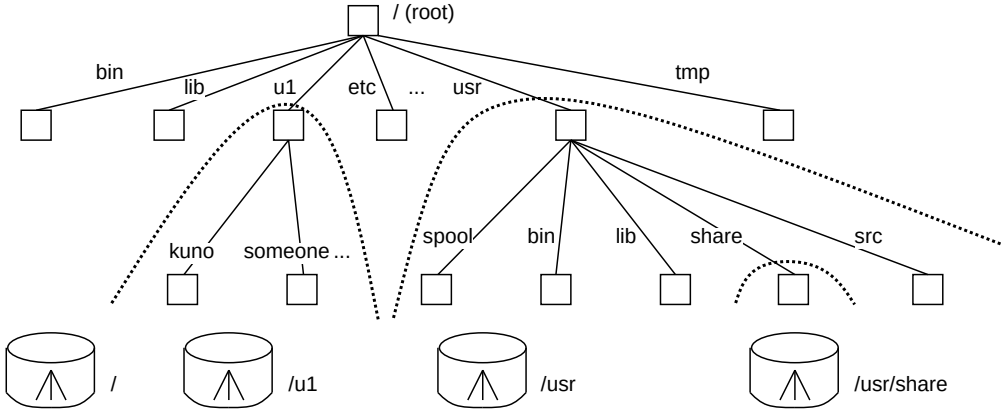

図 11: ディスク装置との対応関係

どのようなディスクがマウントされているかを知るには次のコマンドが利用できます:

- `mount` — ディスクのマウントの状況を表示する
- `df` — 併せて、ディスクの容量、現在使用量を表示する

5.5 ディレクトリの中身は?

先に i-node の説明のところで、ディレクトリとはファイル名と i-番号の対応表だと説明しました。ではディレクトリ自身はどこにどうやって格納されているのでしょうか? それは次の通りです:

[8]サーバによっては各ユーザとして実行される場合も。

[9]Windows 文明ではこれと対照的に、ドライブレターでディスクを明示します。これでひどい目にあった人はいませんか? ディスクを増設するたびに D: E: F: …と増えていって、どこに何があるか覚えておかなければならないという…。

- ディレクトリも普通のファイルと同じ構造を持ち、i-node とそこから指されているデータブロックでできている。名前とその名前に対応する i-番号の対応データは、データブロックに書かれている。
- なので、ディレクトリには別のディレクトリの名前も登録することができる。これがサブディレクトリ。

ルートディレクトリは決まった i-番号「2」を持っていて、Unix の内部ではここからディレクトリを「順番に」たどることで、パス名に対応する任意のファイルを捜すことができます。パス名を指定するたびに、こんなにいくつもディレクトリを捜すのではのろくて役に立たないと思いますか? 確かに速くはないけれど、ディレクトリ 5 段で 5 回たどるのは前にあげた「ディスクを先頭から捜す」のに比べればお話にならないくらい速いわけですし、最近たどった結果をいくつか主記憶に保持しておくことで、何回も同じディレクトリを参照する場合は速やかに捜せることになります。

ここまでずっと、ファイル名やディレクトリ名を○や□の上ではなく、それらを結ぶ線の上に描いてきたことにお気づきでしょうか。つまり、これらの名前はファイルやディレクトリについているのではなく、それに向かってたどるリンクについているというのが真実なのです (普段はあまり意識する必要はありませんが)。また、「.」とか「..」は自分自身、および親へのリンク (そのディレクトリ自身や親ディレクトリの i-番号が取り出せる) ということになります。

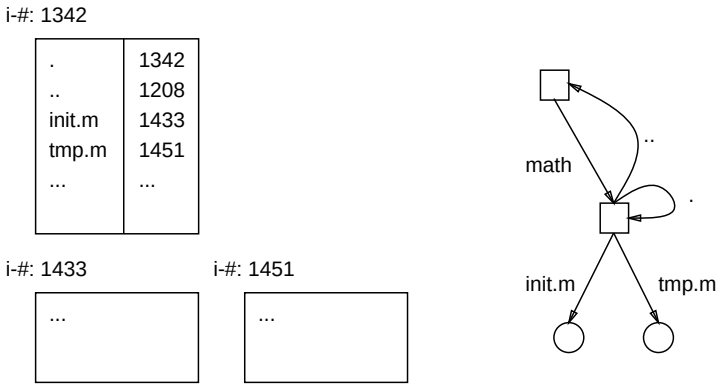

図 12: ディレクトリの中身

ここまで来てようやく、`ln` や `rm` や `mv` の意味がちゃんと説明できます。図 13 にあるように、`ln` というのは既にあるファイルを指すリンクを新しく余計に作る、という意味になります。また、`rm` はリンクを切るコマンドですが、もしその結果ファイルを指すリンクが一つもなくなればそのファイルは本当に削除され、その領域は回収されます。そして `mv` は新しいリンクを作ったあとで古いリンクを消すので図の A と B が同じディレクトリなら結果的に名前が変更されたことになります。また、違うディレクトリであればファイルの位置が移ったことになります。`mv` では同様にしてディレクトリの名前や位置を変更することもできますが、これは `ln`+`rm` ではできません。なぜなら、Unix ではディレクトリに複数名前をつけることは (混乱を避けるため) 許されないようになっているからです。

5.6 シンボリックリンク option

ところで、先にも述べたようにリンクはディレクトリに対して張ることはできません。さらに、普通のファイルでもファイルシステムが違う場合にはリンクを張ることができません。こういう制限はもっともなことではありますが、不便でもあります。[10]

[10] ディレクトリにリンクが張れるとファイルシステムの構造が複雑になります。また、リンクは i-番号によってファイル本体を指し示しますが、i-番号は 1 つのファイルシステム内での固有番号なので、他のファイルシステムのリンクを作ることはできないわけです。

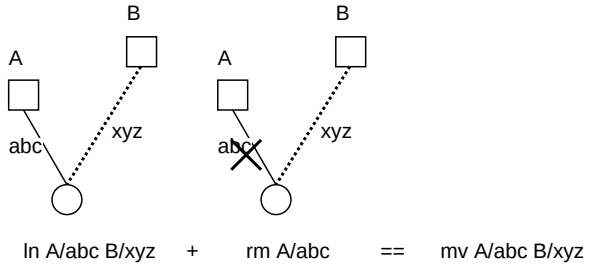

図 13: ln と rm と mv の関係

そこで現在の Unix では、これに加えて**シンボリックリンク**と呼ばれるものも使えるようになっています。

- `ln -s` もとのパス名 新しいパス名 — シンボリックリンクを張る

シンボリックリンクはリンクと同じ `ln` コマンドを使って作成し、意味も類似しています。ただし、シンボリックリンクは「行き先の名前を覚えている」だけで、そこをたどろうとすると行き先の名前に「ジャンプする」仕組みになっているのです。このため、シンボリックリンクはディレクトリでもファイルでも指すことができますし、ファイルシステムの境界に関係なく使うことができます。

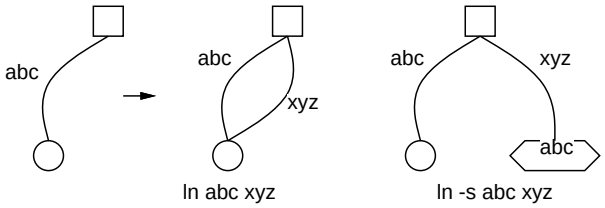

図 14: シンボリックリンクの概念

5.7 ディレクトリ単位の操作 option

`mv` はこれまで「ファイルの名前を変更するコマンド」と説明してきましたが、正確には「これまでの名前を削除し、新しい名前を登録する」ことでこれを実現しています。そういう意味では「`rm` と `ln` を組み合わせたもの」ということになります (ただし、削除と登録は同時に行なわれますから、削除した瞬間にコマンドを殺すとファイルがなくなる、という心配はありません)。

そして、名前を削除するディレクトリと新しい名前を登録するディレクトリは同じでなくてもいいので、これを利用すれば「ファイルをあるディレクトリから別のディレクトリに移動する」ことができます。さらに、ディレクトリに対してはコマンド `rm` や `ln` は使えませんが、`mv` は使えるので「ディレクトリをそっくり別の場所に移す」こともできます。

これらの機能は本来ファイルシステムをまたがっては使うことができないものでしたが、それでは不便なので最近の Unix ではファイルシステムをまたがっての `mv` は `cp` と `rm` の組み合わせでやってくれるようになっています (ただし所要時間はずっと多くかかります)。なお、ディレクトリの (部分) 木構造全体をそのままコピーしたり削除するのは次のコマンドでできます:

- `cp -r` ディレクトリ 行き先 — 指定ディレクトリ以下をコピー
- `rm -r` ディレクトリ — 指定ディレクトリ以下を削除

間違って自分自身の下にコピーしようとすると無限コピーが始まるので注意してください。

5.8 領域管理、探索 option

自分のファイルが増えてくると、その管理も大変になります。まず、自分がどれだけファイル領域を使用しているかを知るにはコマンド **du** を使います:

- `du` ディレクトリ — ディレクトリ以下にあるファイル量合計を示す
- `du -a` ディレクトリ — 個々のファイル名と大きさも表示

一方、様々な条件を指定してそれに合致するファイルを木構造の中で探してくれるのがコマンド **find** です。これには色々なオプションがありますが、いくつかの例を挙げておきます。

- `find` ディレクトリ `-mtime` N `-print` — N 日前に変更したものを探す
- `find` ディレクトリ `-size` N`c -print` — 大きさ N バイトのものを探す
- `find` ディレクトリ `-type d -print` — ディレクトリをすべて打ち出す

ところで、`du` や `find` の出力を見ると、ファイルの表示順が `ls` と違っていることに気がつくはずです。`ls` で普通にファイル一覧を表示するときは abc 順で出て来ますが、これは `ls` がそのように並べ替えて表示してくれているからです。これに対し、`du` や `find` を使うと並べ替える前の (ディレクトリに登録されているままの) 順で表示が行なわれるのです。

6 ファイルの中身

6.1 中身

ファイルには、中身があります。当たり前だと思いますか? 中身のない (長さが 0 の) ファイルに利用価値はないでしょうか? 実はそうでもありません。たとえば Unix では伝統的に、あるファイルが「ある」か「ない」かによって特定の機器が使用中かどうかを表す、という方法が使われて来ましたが、そのような場合、ファイルの中身は不要ですから長さが 0 のファイルを使うのが普通です。

では次にファイルの長さが 0 でないものとして、中身とは何でしょうか? ファイルも計算機のための記憶手段ですから、主記憶と同様、任意のビットの列を (バイト単位で) 格納できます。ファイルの中身をバイトの列として調べるには、`hd` コマンドが利用できます。

- `hd` ファイル — ファイルの内容を 16 進で表示

例を見ましょう。

```
% hd t.c
00000000  6d 61 69 6e 28 29 20 7b  0a 20 20 70 75 74 73 28  |main() {.  puts(|
00000010  22 48 65 6c 6c 6f 2e 22  29 3b 0a 7d 0a           |"Hello.");.}.|
0000001d
%
```

一番左はその行がファイルの何バイト目以降かを 16 進で表示します。次に中央部分で、1 行につき最大 16 バイトずつ、各バイトの値を 16 進 2 桁で表示します。最後に右端には、そのバイトが ASCII の図形文字 (普通に見える文字) の場合、その文字を表示します (図形文字でない場合は代わりに「.」を表示します)。こういう表示を計算機業界では「ダンプ」と呼びます。ダンプカーの「ダンプ」ですね。[11] これを見ると、確かに改行文字 (`0a`) が 1 バイトずつを占めているのが分かります。なお繰り返しになりますが、ファイルの中身は (OS にとっては)「単にバイトの列」であり、それらのバイトの列が何を意味しているかを管理／把握するのは、使う人間の役割です。

では以下で、ファイルやファイルシステムの様々な側面について見てみましょう。

[11] 英語では「吐き出す」という意味。

6.2 種類

使う人にとっては、ファイルには様々な種類があります。普段おなじみなのは「文字が入ったファイル」で、これを**テキストファイル**と呼びます。たとえば次のようなものがそうです:

- 各種プログラム言語ソース
- 電子メールのメッセージ
- その他テキストデータ

これと対照的に、文字として読めないデータが入ったファイルを**バイナリファイル**と呼びます。たとえば次のものがそうです:

- 機械語プログラム
- サウンドデータ、多くの画像データ
- その他バイナリデータ

しかし、「中身」の所に書いたように、Unix にとってはどれも同じ「バイトの列」であって、互いに区別はありません。ではどうやって区別するのでしょう? それには次の方法があります。

- 名前で区別する (`.txt` →テキスト、`.c` → C プログラム...)
- 中身で区別する (実行形式などは先頭に種別が入っている)
- 自分で覚えておく

結局どこからか先は「自分で覚えておく」ことになります。ところで、ファイルの種類を調べる指令もあります。

- `file` ファイル名 — ファイルの種類を調べる

たとえば次の通り。

```
% file loop.c loop.s a.out
loop.c: ASCII C program text
loop.s: ASCII assembler program text
a.out:  ELF 32-bit LSB executable, Intel 80386, version 1 (FreeBSD),
for FreeBSD 5.0, dynamically linked (uses shared libs), not stripped
%
```

ただし、これも名前や中身から「推定」しているだけであり、たまに間違うこともあります。

6.3 テキストファイルとコード系

ファイルの中でも、文字を格納してあるファイルをテキストファイルと呼ぶことは既に説明しました。テキストファイルは**テキストエディタ**と呼ばれるアプリケーションを使って入力したり編集できます。ところで、ファイルに格納されるのはビットの列だったはずですが、どうやって文字が格納できるのでしょう?

答えは、いくつか決まった長さのビットごとに、そのビットがどういうパターンだったらどんな文字、という対応関係をつけておく、ということです。より厳密には、まず文字にそれぞれ固有の番号を振り (この番号を**文字コード**と言います)、次にその番号をビット列に対応づけます (これを**符号化**と言います)。英数字記号に対しては ASCII と呼ばれる文字コード (+符号化) が広く使われていますが、大型機の世界では EBCDIC という文字コードも使われます。これらでは 1 文字を 1 バイトに収

	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	**0**	**@**	**P**	**‘**	**p**
1	SOH	DC1	**!**	**1**	**A**	**Q**	**a**	**q**
2	STX	DC2	**"**	**2**	**B**	**R**	**b**	**r**
3	ETX	DC3	**#**	**3**	**C**	**S**	**c**	**s**
4	EOT	DC4	**$**	**4**	**D**	**T**	**d**	**t**
5	ENQ	NAK	**%**	**5**	**E**	**U**	**e**	**u**
6	ACK	SYN	**&**	**6**	**F**	**V**	**f**	**v**
7	BEL	ETB	**’**	**7**	**G**	**W**	**g**	**w**
8	BS	CAN	**(**	**8**	**H**	**X**	**h**	**x**
9	HT	EM	**)**	**9**	**I**	**Y**	**i**	**y**
A	NL	SUB	*****	**:**	**J**	**Z**	**j**	**z**
B	VT	ESC	**+**	**;**	**K**	**[**	**k**	**{**
C	NP	FS	**,**	**<**	**L**	****	**l**	**|**
D	CR	GS	**-**	**=**	**M**	**]**	**m**	**}**
E	SO	RS	**.**	**>**	**N**	**^**	**n**	**~**
F	SI	US	**/**	**?**	**O**	**_**	**o**	DEL

図 15: ASCII コード表

まる 7 ビットに対応させています。Unix は ASCII 文明に属しています。図 15 に ASCII のコード表を示しておきます。

なお、コードは 16 進表記です。最初の方には 2 文字や 3 文字の名前が並んでいますが、この辺は**制御文字**と呼ばれるもので、行の切れ目を表したりページ替えを表すなど、それぞれ特別な機能を持っています。

ともかく、コード表があればビットの列からそれが何の文字かを知ることができます。次のファイルは何と書かれているでしょうか?

```
% hd t3
00000000  48 6f 77 20 61 72 65 20  79 6f 75 3f 0a     |(ネタバレ省略)|
0000000d
%
```

なお、コード表を引いてみると分かりますが、「0a」というのがこの章のはじめの方でみた改行文字のコードになっています。

6.4 日本語のコード系

さて、日本語を扱う場合はどうでしょうか。8 ビットに収まる文字の数は 256 種類が最大ですから、漢字には全然足りません。そこで漢字を扱うには、2 バイト (16 ビット) を 1 文字に対応させたコード系を使います。

一番基本となる日本語 (16 ビット) の文字コードは JIS 規格のうち **JIS X0208** と呼ばれる規格番号で定められているものです。この規格に掲載されている標準的な符号化では、たとえば、16 進表記で「2121」というパターンの 16 ビットが、文字「！」に対応しています。

ところで、1 つのファイルの中で 8 ビットの文字と 16 ビットの文字を混在させることも普通に行われます。この時は、特別な目印となるコードで「ここから漢字」「ここから ASCII」という切替えを行います。具体例で見てみましょう。

```
% cat t.txt
1．ファイルの一覧を
  表示するには ls を使う。
% hd t.txt
```

```
00000000  31 2e 20 1b 24 42 25 55  25 21 25 24 25 6b 24 4e  |1. .$B%U%!%$%k$N|
00000010  30 6c 4d 77 24 72 1b 28  42 0a 20 20 1b 24 42 49  |0lMw$r.(B.  .$BI|
00000020  3d 3c 28 24 39 24 6b 24  4b 24 4f 1b 28 42 6c 73  |=<($9$k$K$O.(Bls|
00000030  1b 24 42 24 72 3b 48 24  26 21 23 1b 28 42 0a     |.$B$r;H$&!#.(B.|
0000003f
```

つまり ESC-$-B(1b 24 42) が「ここから漢字」、ESC-(-B(1b 28 42) が「ここから ASCII」になっています。この 3 バイトの列の割り当て方は **ISO** (国際標準化機構) によって管理されています。このような形での切替えを含めた符号化を **iso-2022-jp** と呼びます。

ところで、漢字に出入りするたびに 3 バイト費やすのはいやだ、という意見もあります。そこで、ASCII では 8 ビットのうち頭の 1 ビットは常に 0 であることを利用して、漢字は JIS コードを頭のビットのみ 1 に変更したもので表すことで両者を区別する、という方法も使われます。この符号化を **euc-jp**(日本語 EUC) と呼びます。

```
% nkf -e t.txt >t.euc
% hd t.euc
00000000  31 2e 20 a5 d5 a5 a1 a5  a4 a5 eb a4 ce b0 ec cd  |1. ............|
00000010  f7 a4 f2 0a 20 20 c9 bd  bc a8 a4 b9 a4 eb a4 cb  |....  .........|
00000020  a4 cf 6c 73 a4 f2 bb c8  a4 a6 a1 a3 0a           |..ls.........|
0000002d
```

さらに困ったことに、PC の世界では **shift-jis** と呼ばれる符号化方法が使われています。これは EUC をさらにあちこちずらして **8 ビットカタカナ**が使えるようにしたものです。

```
% nkf -s t.txt >t.sj
% hd t.sj
00000000  31 2e 20 83 74 83 40 83  43 83 8b 82 cc 88 ea 97  |1. .t.@.C.......|
00000010  97 82 f0 0a 20 20 95 5c  8e a6 82 b7 82 e9 82 c9  |....  .\........|
00000020  82 cd 6c 73 82 f0 8e 67  82 a4 81 42 0a           |..ls...g...B.|
0000002d
```

そして最近さらに、**UNICODE** と呼ばれるコード系も使われるようになってきました。これは iso-2022-jp のような「切替え」にたよらず、世界中の文字を集めて 1 つの文字コードとしたものです。このため、文字の並び順が JIS と全く違っていて、相互変換には変換表が必要です。[12]UNICODE のテキストファイルでは、**utf-8** と呼ばれる符号化方式が多く使われます。

```
% nkf -w t.txt >t.utf
% hd t.utf
00000000  31 2e 20 e3 83 95 e3 82  a1 e3 82 a4 e3 83 ab e3  |1. ............|
00000010  81 ae e4 b8 80 e8 a6 a7  e3 82 92 0a 20 20 e8 a1  |............  ..|
00000020  a8 e7 a4 ba e3 81 99 e3  82 8b e3 81 ab e3 81 af  |................|
00000030  6c 73 e3 82 92 e4 bd bf  e3 81 86 e3 80 82 0a     |ls............|
0000003f
```

このように、漢字を含むテキストの場合、4 種類の符号化方式が入り乱れているので、注意が必要です。これらの符号化形式の変換は、コマンド **nkf** を使って行うことができます: [13]

- `nkf` ファイル — iso-2022-jp に変換
- `nkf -e` ファイル — euc-jp に変換

[12]iso-2022-jp、euc-jp、sjis の 3 つは 2 バイト文字の中の並び順は同じで、計算で相互に変換できます。
[13]入力ファイルの文字コードは指定しなくても自動的に判別してくれます。

- `nkf -s` ファイル — shift-jis に変換
- `nkf -w` ファイル — utf-8 に変換

また、Unix のようにファイルとは単に「バイト列」を納めたものという扱い方では、「テキスト」ファイルの場合には「行の区切り」方法の約束も必要です。「行の区切り」には前述の制御コードを使うのが一般的ですが、ここがまたシステムによって微妙に違っています。具体的には、次のような違いがあるので注意が必要です:

```
Unix    →  NL
Windows →  CR NL
Mac     →  CR
```

6.5 文字コードを直接扱うプログラム

上に示したように、テキストファイルとは単に「文字コードとして正しいバイトだけを含んでいるファイル」なわけで、プログラムから何かを出力するときに「文字用の出力」と「バイナリデータの出力」が区別されるわけでは (あまり) ありません。たとえば、ASCII 文字コードを順番に出力するプログラムというのを見てみましょう:

```
/* genascii.c -- generate ASCII chars */
main() {
  int i = 32; /* ASCII SP */
  while(i < 127) { /* ASCII DEL */
    putchar(i); ++i;
  }
  putchar(10); /* ASCII NL */
}
```

これを動かすと次のようになります。

```
% gcc t05.c
% a.out
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXY
Z[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
%
```

確かに ASCII コードの文字が順番に出力されています。

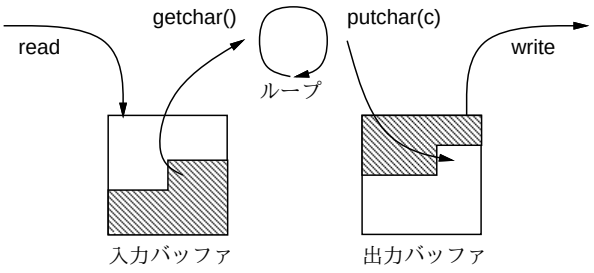

図 16: バッファつき入出力

ところで、この `putchar()` はさっきの `write()` とどう違うのでしょうか? `write()` はシステムコールなので、1 回呼び出すにつき 1 回、OS の中を呼び出しますが、それには結構な手間が掛かります。それに対し、`putchar()` は渡された文字を 1 文字ずつ**バッファ**(ためておく場所) に蓄積していき、満

杯になったら write() で書き出してまたバッファの最初から 1 文字ずつため始めます。(図 16)。こうすればばシステムコールが頻繁に起こることはなくなります。このような入出力機構をバッファつき入出力と呼びます。

ここでようやく、前の章で使った printf の説明ができます。printf はちょっとしたメッセージを書き出すためのもので、内部では putchar(と同等のもの) を使用している、バッファつき入出力サブルーチンの 1 つだったわけです。

7 まとめと演習

この章では、計算機システムで恒久的にデータを保管する仕組みであるファイルシステムの原理やさまざまな側面について、ひととおり学びました。自分の「大切な」データはファイルシステムに保管してもらうわけですから、その機能を理解して使いこなすことは重要ですし、よい「計算機生活」を送る秘訣だとも言えますね。

3-1. ファイルのモードをいろいろ変更し、ls で表示させてみて、正しく変更されているか確認しなさい。特に「自分に読めない」「自分に書けない」などの設定が確かにそのように働いているか確認してみなさい。また、友人に協力してもらい、「誰にでも読める」「誰にでも書ける」にしたファイルを他人が読み書きできることを確認しなさい。

3-2. 自分のホームディレクトリの下にサブディレクトリを作り、中にいくつかファイルを用意した上でそのサブディレクトリのモードをいろいろ変更してみなさい。ディレクトリの「読めない」「書けない」「実行できない」という保護はそれぞれどんな意味を持ちますか? その中の個別のファイルに対する保護とはどう違うでしょうか? また、友人に協力してもらい、「誰にでも読み書きできる」ディレクトリの下にあるファイルを消せるかどうか、またそこに新しいファイルを作れるかどうか、その作ったファイルをあなたが読み書きできるかどうか等を試しなさい。

3-3. largefile.c を打ち込んでコンパイルし、自分のホームディレクトリで動かし書き込みスループットを計測してみなさい。続いて、サーバ sma に行って同じことをやってみなさい。その違いは何から来るのか説明してみなさい。他にやってみたければ、各 PC のローカルディスクに作る (ファイルを/var/tmp/test などとする)、各 PC のメモリファイルに作る (ファイルを/tmp/test などとする)、なども比較してみなさい。

3-4. largifile.c を直して、ファイルを 1 バイトずつ write するように直して 100 万回ループさせ、システムコールの 1 回あたりに掛かる時間を計測し、通常の命令 (1 回目にやりましたね) の何命令ぶんくらいの時間が掛かるかを計算してみなさい。[14]

3-5. genascii.c を参考にして、自分の名前を漢字で打ち出すプログラムを作ってみなさい。または、JIS 漢字コードの表を「見やすく」生成するプログラムを作ってみなさい。ただしプログラム中に直接日本語の文字を入れないこと。出力コードは JIS、EUC、SJIS のどれでも構いません。[15]

3-6. 英数字、ひらかな、漢字の入った短いファイル (たとえば、「12345 あいうえお ABCDE 亜伊宇江尾」とか) を作り、それを iso-2022-jp のほかに euc-jp、sjis、utf-8 にも変換してみてそれぞれ hd コマンドで打ち出し、コードの対応関係がおおよそどのようになっているか検討しなさい。

[14] 100 万バイトの出力のためにディスクのアクセス時間は無視できると考えてよいでしょう。
[15] ヒント: JIS コードの文字は 2 バイトで、1 バイト目も 2 バイト目も 16 進数で 21〜7E の範囲に割り当てられています。