第3章　ファイルシステムとデータ記憶

大容量かつ安定したデータ記憶を提供する「ファイルシステム」は計算機システムの中でもとても重要な位置を占める。というのは、利用者が操作したり保管しておく情報は (プログラムによって加工されている瞬間をのぞいては) 常にファイルシステムによって管理されているからである。本章では Unix を題材としてファイルシステムや入出力メカニズムの概要について説明する。

3.1　2次記憶とファイルシステム

メモリは主記憶装置とも言われる通り、計算機システムにおける「主要な」記憶装置であるが、電源を切ると記憶内容が消えてしまう、またビットあたりコストが高いためそう大量に備えることができないなどの弱点を持っている。

そこで、ビットあたりコストが低く電源を切っても消えないような記憶装置を計算機システムの「2 次記憶装置」として使用する。現在のところ、安定性と価格の両面からもっとも広く使われている 2 次記憶装置は磁気ディスク装置である。その原理を図 3.1 に示す。

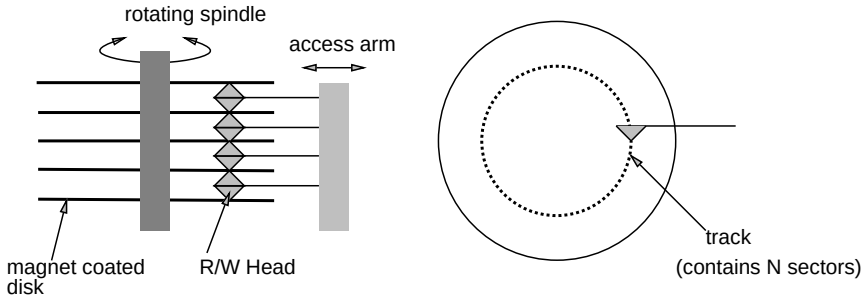

図 3.1: 磁気ディスク装置の構造

すなわち、回転する円盤の表面に (カセットテープ同様) 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記録する。記録はらせん状ではなく同心円状で、その 1 つの同心円を「トラック」と呼ぶ。ヘッドが固定

されている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る。また、トラック内のデータブロック (通常 512 バイト、可変のものもある) を「セクタ」と呼ぶ。なお、フロッピーディスクというのはこれの廉価盤で、円盤がぺらぺらの材質で 1 枚だけケースに入っているものである。

ところで、これにデータを読み書きするには、「どのトラックのどのセクタを読め/書け」と CPU が言えばいい。さて、そういうのが裸で計算機とつながっていたら、あなたは幸せか? もちろん、幸せでないから OS が手助けしてくれるのだが。OS の中でも「ファイルシステム」とか「データ管理」とか呼ばれる部分はおおむね次のようなことをやってくれる。

- 「どれだけの領域が欲しい」「返却する」という要求に対処。
- 他の人と領域がかちあわないように管理。
- 他の人のデータを壊したり、他人に勝手に読まれないよう管理。
- トラック番号、セクタ番号でなく名前で指定できるようにする。
- 名前も他人とぶつかったりしないような機構を用意する。
- 実際にデータを読み書きする手助け。

これも見かたによれば「ディスク上の領域」という「資源」を管理していることになるわけである。

3.2　ファイルとその属性

3.2.1　ファイルとは?

すでにご存知の通り、利用者はディスク上に置かれるデータの集まりを「ファイル」という概念で捉える。では、ファイルとは要するにどんなものか? 具体的には次のような特性が重要であろう。

- 情報を蓄えておく場所/容れ物である。
- 長期的、恒久的に記憶できる。
- データ本体にいくつかの情報 (たとえば名前) を付随させられる。

単なるセクタの並びでしかないはずのディスク装置が、ファイルシステムより上のレベルから見ると、「ファイルの集まり」であるかのように見える。これはちょうど 1 個しかない CPU とのっぺらぼうの主記憶の上に「プロセス」の集まりがあるかのように見えるのと同様である。プロセスもファイルも現実には存在しないが OS によって「あたかもあるかのように」扱うことができる、計算機固有の「概念」である。さて、以下ではファイルの様々な側面について見てみよう。

3.2.2 名前

ファイルには、名前がついている (名前の長さ?使える文字?)。ファイルの名前を調べるには、ls 指令を使う。ただの ls では名前の最初が「.」で始まるものは表示されず、それらも表示させたい場合には-a オプションを指定する:

```
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   -- 加えて、「.」で始まるファイルも表示
```

例えば次の通り。

```
% ls
Library MH      Mail    t00.c   t00.s
% ls -a
.               .bashrc         .logout         .profile        Mail
..              .canna          .mh_profile     .twmrc          t00.c
.NeXT           .cshrc          .mule           .x11defaults    t00.s
.Xauthority     .elm            .newsrc         .xinitrc
.bash_history   .history        .newsrc.bak     Library
.bash_profile   .login          .nn             MH
```

「.」で始まるファイルが多いのに驚かれましたか? 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る。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だった。

3.2.3 長さ

ファイルには、長さがある。それは当たり前だと思いますか? 長さはあるのだけれど、正確にはわからないという OS も沢山ある。Unix ではファイルの長さはバイト単位で簡単に調べられる。それには

```
ls -l   -- 長さを始めとする詳しい情報を表示。
```

によればよい。ところで、

```
% cat t.c
main() {
  puts("Hello.");
}
%
```

というプログラムを入れたファイルがあったとする。そのファイルの長さはいくつだと思うか?

```
% ls -l t.c
-rw-------  1 someone        29 Apr 22 13:49 t.c
%
```

3.2.4　中身

ファイルには、中身がある。(当たり前だと思いますか? 中身のないファイルに利用価値はないか?) 中身とは何か? 計算機のための記憶だから主記憶と同様、任意のもビットの列を (バイト単位で) 格納することができる。ファイルの中身をバイトの列として調べるには、前回もやったように od コマンドを使う:

```
od -x ファイル名    -- 16 進形式で表示
od -c ファイル名    -- 文字形式で表示
```

なお、こういう表示を計算機業界では「ダンプ」と呼んでいる。日本語に直すと「オミヤゲ」だが…[1]

```
% od -x t.c
0000000 6d61 696e 2829 207b 0a20 2070 7574 7328
0000020 2248 656c 6c6f 2e22 293b 0a7d 0a00
0000035
% od -c t.c
0000000   m   a   i   n   (   )       {  \n           p   u   t   s   (
0000020   "   H   e   l   l   o   .   "   )   ;  \n   }  \n
0000035
%
```

3.2.5　種類

使う人にとっては、ファイルには様々な種類がある。

- テキストファイル (お手紙、プログラム、データ、...)
- 実行形式ファイル
- オブジェクトファイル
- ライブラリファイル

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、互いに区別はない。ではどうやって区別するかというと。

- 名前で区別する。(.p → Pascal プログラム、.c → C プログラム...)

[1]「吐き出したもの」のことだから。

- 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」ことになる。ところで、ファイルの種類を調べる指令もある。

```
file ファイル名...   -- ファイルの種類を調べる
```

ただし、これも名前や中身から「想像」するだけ間違う場合もある。

```
% file t00.c t00.s a.out
t00.c:          c program text
t00.s:          [nt]roff, tbl, or eqn input text
a.out:          ELF 32-bit MSB executable SPARC Version 1, ...
%
```

どうやら file はアセンブリ言語を知らないらしい。

3.2.6 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から、その情報もあることは分かる。ほかに、最後に読み出した日時も記録されている。

```
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、変更日時の変わりに読み出し日時を使う
```

なお、ls ではこれらを自由に組み合わせていい。たとえば-a と-l と-t をまとめて指定して ls -lat というのも可能。ついでに、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る。

```
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、変更日時の変わりに読み出し日時を使う
```

たとえば次の通り。

```
% ls -l
total 14                                通常は ABC 順↓
drwxrwsr-x  4 someone       512 Apr 18 16:43 Library
drwxr-xr-x  4 someone       512 Apr  7 16:51 MH
drwx------  2 someone       512 Apr  1 16:55 Mail
-rwx------  1 someone      7456 Apr 22 13:49 a.out
-rw-------  1 someone        29 Apr 22 13:49 t.c
-rw-------  1 someone       122 Apr 22 13:42 t00.c
```

```
-rw-------  1 someone       482 Apr 22 13:42 t00.s
% ls -t    時刻順↓
a.out   t.c     t00.s   t00.c   Library MH      Mail
% ls -lu
total 14                    読み出し時刻順になった↓
drwxrwsr-x  4 someone       512 Apr 22 13:41 Library
drwxr-xr-x  4 someone       512 Apr 22 13:41 MH
drwx------  2 someone       512 Apr 22 03:17 Mail
-rwx------  1 someone      7456 Apr 22 13:58 a.out
-rw-------  1 someone        29 Apr 22 13:51 t.c
-rw-------  1 someone       122 Apr 22 13:58 t00.c
-rw-------  1 someone       482 Apr 22 13:58 t00.s
%
```

3.2.7　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) も表示される。なぜ、持ち主の情報が必要か? 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のは持ち主が分からなければ不可能だから。さらに、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」というものが提供されている。ファイルには持ち主に加えて所属グループの情報が付随している。それは

`ls -lg`　-- `-l` と同じだが、所属グループも表示

によって見ることができる。また、各ユーザは 1 つ以上のグループに所属している。ファイルの持ち主は

`chgrp` グループ名 ファイル …

によりファイルの所属グループを設定できる。

```
% ls -lg t1              ↓私 (久野) は教官グループの人
-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の人でもある。
-rw-r--r--  1 kuno     wheel         894 Apr 22 14:03 t1
%
```

なお、皆様は 1 つのグループにしか所属していません。(何というグループでしょうか?)

3.2.8 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ぶ。具体的なモードとしては各ファイルごとに

```
user(持ち主)     \       / read(読める)       \
group(グループ)  -ごとに- write(書ける)      -をそれぞれ設定
other(その他の人) /     \ execute(実行できる) /
```

できる。このモードは `ls -l` の表示の最初の部分に含まれている。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る。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だが、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る。

モードを変更するのは次による:

```
chmod [u][g][o](+|-)[r][w][x] ファイル …    -- モード設定
```

例えばさっきのファイルでやってみる。

```
% 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
```

3.2.9 i-番号

i-番号というのはファイル一つ一つにつく固有番号である (PID のようなものと思えば良い)。`ls -i` でこれを表示させることができる。なぜ名前があるのにそんなものが必要なのか? 実は、名前は変化することがある。また、一つのファイルに別の名前をつけることもできる。

```
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
```

実際やってみよう。

```
% 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 はファイルを消すのではない! ただ、全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るわけである。そして、「全ての名前が無効になった」ことは各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くことで知ることができる。

3.2.10　その他の良く使う指令

ここまでに出てこなかったがファイルに関連してよく使う指令のリストを掲げておく。

```
wc ファイル...                -- 字数、語数、行数を数える
grep パターン ファイル...     -- パターン探索
head ファイル                 -- ファイルの先頭部分のみ表示
tail ファイル                 -- ファイルの終り部分のみ表示
cp ファイル1 ファイル2        -- ファイルのコピー
diff ファイル1 ファイル2      -- 2つのファイルの違いを表示
cat ファイル...               -- ファイルの中身を画面に表示
less ファイル...              -- 1画面ずつ止まりながら表示
```

すこし活用してみよう。

```
% gcc -S t00.c
% wc t00.s
      37      64     482 t00.s  ←行数は37行
% head -5 t00.s                 ←最初の5行
        .file   "t00.c"
.section        ".rodata"
        .align 8
.LLC0:
        .asciz  "Hello.\n"
% tail -4 t00.s                 ←最後の4行
        restore
.LLfe1:
        .size    main,.LLfe1-main
        .ident  "GCC: (GNU) 2.5.7"
% cp t00.c t00a.c               ←コピー
% mule t00a.c                   (直す)
...
% gcc -S t00a.c
% diff t00.s t00a.s             ←違いは?
1c1
<       .file   "t00.c"
---
>       .file   "t00a.c"
5c5
<       .asciz  "Hello.\n"
---
>       .asciz  "Bye.\n"
```

なお、cat と less はやるまでもないですね?

表 3.1: ASCII コード表

00	NUL	01	SOH	02	STX	03	ETX	04	EOT	05	ENQ	06	ACK	07	BEL
08	BS	09	HT	0A	NL	0B	VT	0C	NP	0D	CR	0E	SO	0F	SI
10	DLE	11	DC1	12	DC2	13	DC3	14	DC4	15	NAK	16	SYN	17	ETB
18	CAN	19	EM	1A	SUB	1B	ESC	1C	FS	1D	GS	1E	RS	1F	US
20	SP	21	!	22	"	23	#	24	$	25	%	26	&	27	’
28	(| 29 |)	2A	*	2B	+	2C	,	2D	-	2E	.	2F	/
30	0	31	1	32	2	33	3	34	4	35	5	36	6	37	7
38	8	39	9	3A	:	3B	;	3C	<	3D	=	3E	>	3F	?
40	@	41	A	42	B	43	C	44	D	45	E	46	F	47	G
48	H	49	I	4A	J	4B	K	4C	L	4D	M	4E	N	4F	O
50	P	51	Q	52	R	53	S	54	T	55	U	56	V	57	W
58	X	59	Y	5A	Z	5B	[	5C	\	5D	]	5E	^	5F	_
60	‘	61	a	62	b	63	c	64	d	65	e	66	f	67	g
68	h	69	i	6A	j	6B	k	6C	l	6D	m	6E	n	6F	o
70	p	71	q	72	r	73	s	74	t	75	u	76	v	77	w
78	x	79	y	7A	z	7B	{	7C	|	7D	}	7E	~	7F	DEL

3.3　テキストファイルと文字コード

3.3.1　符号化とコード系

ファイルの中でも、文字ばかりを格納してあるファイルを「テキストファイル」と呼ぶ。テキストファイルがなぜ重要かというと、人間が直接読んだり書いたりできるのは (狭い意味では) テキストファイルだけだからである。これと対比してテキストファイルでないファイルを「バイナリファイル」と呼ぶこともある。テキストファイルを書いたり修正するにはテキストエディタを使う。mule がその例である。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。どうやって文字が格納できるのだろうか?

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、という約束というか対応関係をつけておく、ということである。これを「符合化」という。英数字記号に対しては ASCII と呼ばれる符合化が広く使われている (が、IBM 文明などでは EBCDIC というのもある。) これらでは 1 文字を 8 ビット (=バイト) に対応させている。Unix は ASCII 文明である。表 3.1 に ASCII のコード表を示す。

なお、コードは 16 進表記である。最初の方には 2 文字や 3 文字の名前が並んでいるが、この辺は制御文字であって普通の文字ではない。ともあれ、こ

れ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る。次のファイルは何と書かれているか?

```
% od -x t3
0000000 486f 7720 6172 650a 796f 753f 0a00
0000015
%
```

3.3.2 日本語のコード系

さて、問題は漢字である。8 ビットに入り切る文字の数は 256 種類しかないので、漢字には全然足りない。そこで漢字を扱うには 2 バイトを 1 文字として符合化をする。符合化のしかたは、JIS 規格に従う。ところで、ASCII の文字と JIS 漢字とでは重なる部分があるので (例えば「!"」は何と「、」と同じである)、これらを区別して扱う方法が必要である。我々のところでは、ISO で規定されている 3 バイトの列で「ここから漢字」「ここから ASCII」を指定している。

```
% cat t.txt
1．ファイルの一覧を
  表示するには ls を使う。
% od -x t.txt
0000000  312e 201b 2442 2555 2521 2524 256b 244e
0000020  306c 4d77 2472 1b28 420a 2020 1b24 4249
0000040  3d3c 2824 3924 6b24 4b24 4f1b 2842 6c73
0000060  1b24 4224 723b 4824 2621 231b 2842
0000076
```

つまり ESC-$-B が「ここから漢字」、ESC-(-B が「ここから ASCII」になっている。これが一応由緒正しい JIS に従った符合化である。(ところが、漢字に切り替えるのに ESC-$-@、8 ビット側に戻るのに ESC-(-J や ESC-(-H を出力するアプリケーション、またこれらのコードでないとうまく動かないアプリケーションが存在する。ESC-(-J は JIS8 ビットコードへの切り替えだから「間違い」ではないが、それ以外の 2 つは現在では古くなった規格である。どちらにしろ困った問題ではある。)

ところで、漢字に出入りするたび 3 バイト費やすのはいやだ、という意見もある。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、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せば両者を区別できる。これを EUC と呼んでいる。

```
% nkf -e t.txt >t.euc
% od -x t.euc
```

```
0000000  312e 20a5 d5a5 a1a5 a4a5 eba4 ceb0 eccd
0000020  f7a4 f20a 2020 c9bd bca8 a4b9 a4eb a4cb
0000040  a4cf 6c73 a4f2 bbc8 a4a6 a1a3
0000054
```

さらに困ったことに、パソコンの世界ではシフト JIS なるものが流布している。これは漢字は 1 バイト目のみ頭のビットが立ち、2 バイト目は ASCII の記号の部分をよけるように符合化したものである。

```
% nkf -s t.txt >t.sj
% od -x t.sj
0000000  312e 2083 7483 4083 4383 8b82 cc88 ea97
0000020  9782 f00a 2020 955c 8ea6 82b7 82e9 82c9
0000040  82cd 6c73 82f0 8e67 82a4 8142
0000054
```

このように、漢字を含むテキストの場合 3 種類の符合化がいり乱れているので注意が必要である。とりあえずわれわれのところにシステムは JIS を基本としているが、PC から持ってきたものは SJIS のままであるのが普通なので自分でコード変換しないといけない。既に上で使ってしまったが、コード変換指令をまとめると次の通りである。

```
nkf ファイル >出力ファイル       -- JIS に変換
nkf -e ファイル >出力ファイル    -- EUC に変換
nkf -s ファイル >出力ファイル    -- SJIS に変換
```

3.4　ファイルシステムと名前空間

3.4.1　ディレクトリ

これまで、login したらそこで ls というと自分のファイルが見える、というのを当たり前のように考えてきたが、これはよく考えてみるととても不思議なことではないだろうか? 例えば someone さんが login して自分の仕事をした後で私が同じホストに login して仕事できる、ということから、同じ機械の中に両方の人のファイルがともに存在することは明らかである (ご存じの通り、同時に使うことさえできる)。でも、なぜ二人のファイルはまぜこぜになってしまわないのだろう?

実は、Unix ではファイルはディレクトリ (=登録簿、というような意味) という単位にまとめられて管理されている。ディレクトリにはいくらでもファイルを登録しておくことができる。すべてのプロセスには「現在位置」(カレントディレクトリ) が対応していて、特に指定しなければ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。従って、上の質問の答えは「二人のいるディレクトリが違うから」である。この様子を図 3.2 に示す。なぜ違うか?それは login を担

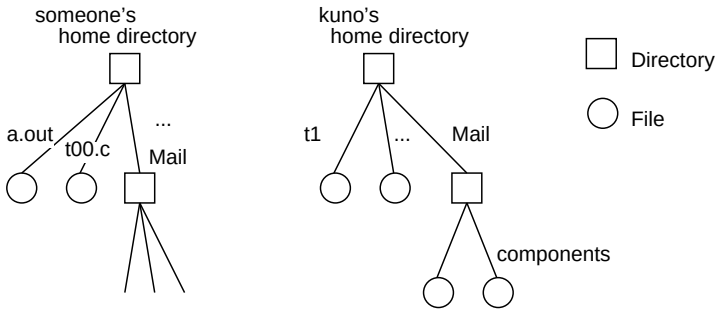

図 3.2: ディレクトリの概念

当する部分の処理が、ユーザ名に応じてそれぞれに固有のディレクトリを現在位置にしてから bash(コマンドインタープリタ) を起動するからである。この、login した時の現在位置を各自の「ホームディレクトリ」とよぶ。

ところで、実は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る。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」という。例えば図 3.2 では 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る。さらにサブサブ、サブサブサブ、… といくら作ってもよい。サブディレクトリを作る/消すには

```
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
```

による。ディレクトリは `ls -l` ではモード表示の最初に「d」と表示されるので分かる。あるいは、`ls -F` による表示では名前のあとに「/」がついて表示される。

3.4.2 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 3.2 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だろう。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」になっていて、全てのディレクトリやファイルはこの「子孫」である。図 3.3 われわれのシステムにおけるディレクトリの木構造の概要を示す。このように、われわれのシステムでは各ユーザのホームディレクトリはルートの直下にある u1〜u3 というディレクトリの下にまとまっている。

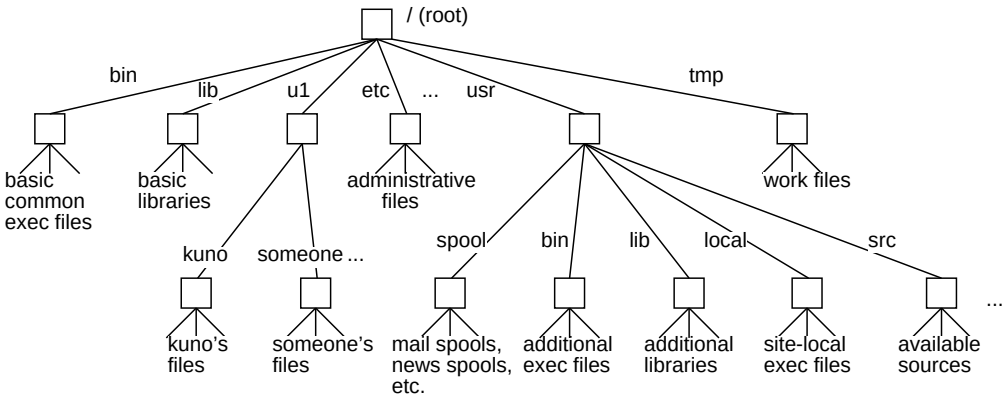

図 3.3: ディレクトリの木構造

3.4.3　パス名

図 3.3 の中には lib とか bin という名前のディレクトリが複数あるし、また各自が同じファイル名を考えつくこともある。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なわけだが、それらを区別して指定する方法が必要である。それには「パス名」というものを使う。パス名には次の 2 通りがある。

```
/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る。例えば

```
/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
```

のような具合である。

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示す。だから、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` を意味することになる。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だ。

ところで、パス名の成分には特別な名前として

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

というのが使える。例えば同じく 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ある。

```
.                       -- 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ある。そのための指令として次のものがある。

```
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、そのコマンドを打つ手間が増える代わりにそこにあるファイルは名前だけで楽に指定できる。例えば次の例を見ていただく。(あと、mv は名前だけでなくディレクトリ位置も移せることに注意。)

```
% mkdir work
% ls
Library MH      Mail    t00.c   t00.s   t00a.c  t00a.s  t3      work    zzz
% cp t00.c work/t00.c
% cp t00a.c work/t00b.c
% mv t3 work/t3
% diff work/t00.c work/t00b.c
6c6
<     printf("Hello.\n");
---
>     printf("Bye.\n");
% cat work/t3
How are
you?
% cd work
% ls
t00.c   t00b.c  t3
% diff t00.c t00b.c
6c6
```

```
<     printf("Hello.\n");
---
>     printf("Bye.\n");
% cat t3
How are
you?
% ls ..
Library MH      Mail    t00.c   t00.s   t00a.c  t00a.s  work    zzz
% cat ../zzz
How are you?
% mv ../zzz t4
% ls
t00.c   t00b.c  t3      t4
% cat t4
How are you?
%
```

3.4.4　デバイスファイル

ところで、ディレクトリの中に格納できるものに、ファイルとディレクトリに加えて「デバイスファイル」と呼ばれるものがある。これは、ディスクやテープ、端末などの入出力装置に対応している。普通の Unix システムでは /dev ディレクトリの下にデバイスファイルを集めておく習慣になっている。

このようにファイル以外のものもディレクトリ階層の中で整理できるようにしておくことは、さまざまな種類のものを統一的に管理できるという利点を提供する。

また、使い方の面から見ても Unix の多くのユティリティにはファイルと入出力装置の両方で動作するように作られたものが多い。たとえばテープにバックアップを取る指令である tar は

`tar cf /dev/rmt8 .`　--- テープにカレントディレクトリ以下をバックアップ

`tar cf ../b.tar .`　--- テープでなくファイルにバックアップを書く

のように使い分けられる (図 3.4。ファイルにバックアップを書くことは、そのファイルをネットワーク経由で転送し、向こう側で再度内容を取り出すことでディレクトリの部分階層一式を簡単に送る方法としてよく使われる。

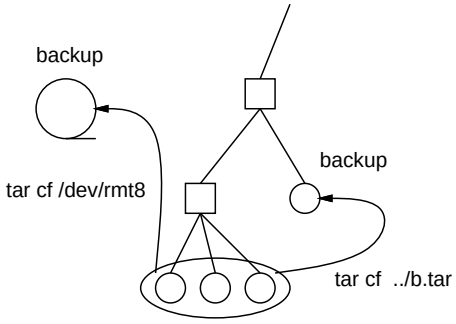

図 3.4: テープへの/ファイルへのバックアップ

3.4.5 名前空間とマウント

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に構成されているのだが、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とうてい不可能であるし、そうしてしまうとネットワーク経由で共有されている部分、自分固有の部分などを自由に混ぜることができない。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る。この様子を図 3.5 示す。ネットワーク共有もこのディスク単位で行なえる。どのようなディ

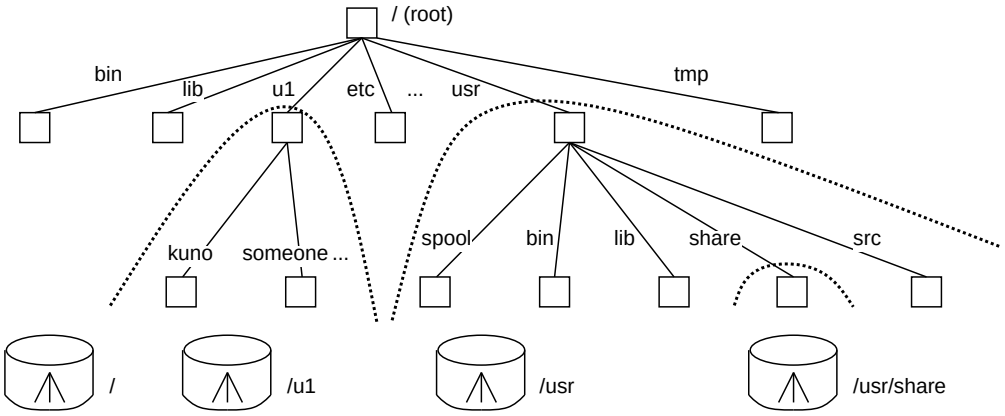

図 3.5: ディスク装置との対応関係

スクがマウントされているかを知るには次のような指令が利用できる。

```
/etc/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
```

3.5　入出力とシステムコール

3.5.1　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やディレクトリがどう見えるかについて説明してきた。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説明する。

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ある。これは、もしそんなことをしたとすれば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うことが簡単に起きるから当然である。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頼まれた操作を行ってくれるわけである。この「OS に作業を頼む」ことをシステムコールと呼ぶ。

Unix で C 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る。入出力のためのシステムコールは次の 2 つである。

```
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る。標準では 0、1、2 の 3 つのディスクリプタが用意される。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す。read ではファイルのおわりになるとそれ以上読めなくなるのでバイト数として 0 が返る。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る。

```
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」である (図 3.6 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ということである (図 3.6 右)。

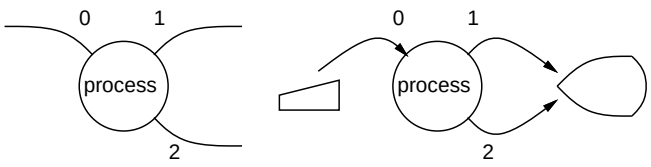

図 3.6: ファイルディスクリプタと入出力

3.5.2 リダイレクトとパイプライン

ではさっそく、簡単なプログラムを示してみよう。次は簡単なメッセージを画面に打ち出すプログラムである。

```
/* t03.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た。この違いについては後で説明する。ともあれ、これを動かしてみる。

```
% gcc t03.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る。

```
% gcc t03.c
% 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る。だから画面には何も現われないが、代わりに出力はファイルに書き込まれている。

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

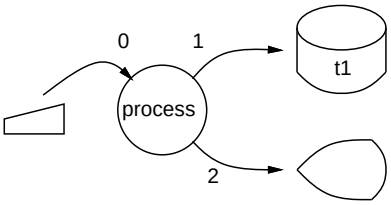

図 3.7: 出力の切り替え

これを Unix では出力のリダイレクトと呼ぶ。この様子を図 3.7 に示す。ところで 2 番 (標準エラー出力) は何のためにあるのだろう? 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ある。図 3.7 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。

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、ということである。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大丈夫なわけである。このような性質を「(入出力の) 機器独立性」という。何がありがたいかわかりますか? 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全部使用する命令が違うとしたらプログラムを書くのは相当煩わしいと思いませんか?

Unix のもう 1 つの特徴として、チャネルへの出力を、別のプロセスの入力に接続できる、という機能がある。これを「パイプライン」と呼んでいる。例えば次の通り。

```
% a.out | tr a-z A-Z
THIS IS A PEN.
%
```

なお「`tr a-z A-Z`」は見ての通り、小文字を対応する大文字に置き換える機能を持つ。この接続の様子を図 3.8 に示す。tr にとっては入力チャネルである 0 番が a.out からの接続に切り替わっていることに注意。また tr の 1 番は特に指定していないので画面のままである。

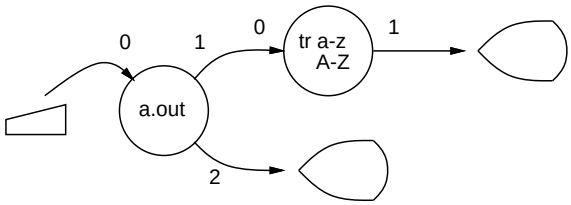

図 3.8: パイプライン接続

3.6　入力と出力

さて、出力はわかったから今度は入力を見てみよう。

```
/* t04.c -- read and write */
```

```
#define BUFSIZE 20
char buf[BUFSIZE];

main() {
  int n;
  n = read(0, buf, BUFSIZE);
  write(1, buf, n);
}
```

このプログラムは最大 20 バイトまで入るバッファを用意し、そこに入力を読み込んでそのまま書き出す。

```
% gcc t04.c
% a.out
This is a pen.
This is a pen.
% a.out
This is a pen.  That is a dog.
This is a pen.  That%  is a dog.
bash: is: command not found
%
```

このように、入力がバッファの長さ以内なら全部がコピーされるが、長すぎる場合には余った入力は読まれない (代わりに bash が読んでしまっている)。では、長いファイルをコピーするにはどうしたらいいだろう? バッファを大きくする? 根本的解決ではありませんね。もちろん、read と write をループにより繰り返し実行するのが普通。

```
/* t05.c -- loop of read and write */

#define BUFSIZE 20
char buf[BUFSIZE];

main() {
  int n;
  while((n = read(0, buf, BUFSIZE)) > 0) {
    write(1, buf, n); }
}
```

なお、この while の条件はえらくごちゃごちゃしているが、「read を呼び出し、返って来た値を変数 l に入れるが、さらにこの入れた値が 0 より大きい間ループを実行し続ける」という意味である。これを使えば長いファイルでもコピーできる。

```
% ls -l t2
-rw-r--r--  1 kuno        10000 Apr 20 11:54 t2
% a.out <t2 >t3
% ls -l t3
-rw-r--r--  1 kuno        10000 Apr 20 11:58 t3
% time a.out <t2 >t3
        0.3 real         0.0 user         0.2 sys
%
```

10000 バイトのファイルをコピーするのに、ユーザ CPU 時間はほとんど 0 だが、入出力では OS が働くわけで、それに 0.2 秒ほど CPU を使っている。

ところで、バッファを大きくするのは根本的解決ではないと上で述べたが、しかし 20 というのが一番よいというわけでもない。たとえば極端な話、上の BUFSIZE を 1 にしてみよう。

```
% time a.out <t2 >t3
        3.7 real         0.2 user         3.4 sys
%
```

10 倍も時間が掛かるようになってしまった! これは、read や write を使うたびにシステムコールが起き、OS に切り替わるのにかなり CPU 時間が掛かるためである。実際、上の値からおおざっぱに見て、read と write のシステムコールを 1 回ずつ実行するのに 340nsec 掛かっているわけで、これは加算命令なら 680 命令も実行できる時間に相当する。だから上のようなケースでは、システムを圧迫しない範囲において大き目の BUFSIZE を使用するのが効率よい。

しかし、プログラムの内容によっては 1 文字ずつ入力を読んだり 1 文字ずつ書き出したりしたい場合もある。そこで、そのような時には read と write の代わりに getchar と putchar を使う。

```
/* t06.c -- loop of getchar and putchar */

main() {
  int c;
  while((c = getchar()) >= 0) {
    putchar(c); }
}
```

なお、getchar はファイルの終りまで読んでこれ以上残った文字がなければ負の数を返す。

```
% gcc t06.c
% time a.out <t2 >t3
```

```
        0.2 real         0.0 user         0.1 sys
%
```

今度は大丈夫である。実は、getchar はまず入力バッファに read でまとめて読み込み、その中から 1 つずつ文字を取り出して返し、バッファがからっぽになったら再度 read で読み込むようなサブルーチンである。同様に、putchar はバッファに 1 文字ずつ溜め込み、バッファが満杯になったらはじめて write で書き出す (図 3.9)。これらを使えばシステムコールが頻繁に起こることはない。このような入出力機構をバッファつき入出力という。

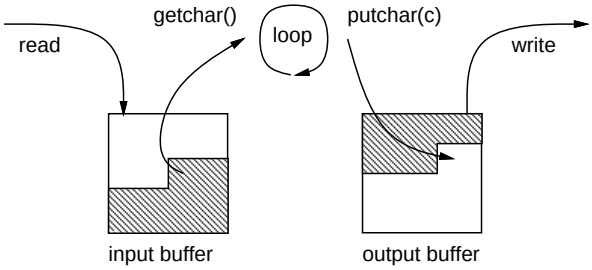

図 3.9: バッファつき入出力

ここでようやく、前回 printf というのを使った理由が説明できる。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 つだったわけである。

3.7 ファイルシステムの内部 — Advanced Topics

3.7.1 再び、ファイルの名前について

ここまでずっと、ファイルに名前という属性がある、という説明をしてきたが、実はそれは嘘である。というのではあんまりだから、もう少しおだやかな質問をしてみよう。ファイルの名前は、どこに格納されていると思いますか?

もちろん、これに答える前にその他のファイルの属性一般はどこに格納されているかを知らないといけない。図 3.10 に示すように、個々のディスクはそれぞれ i-領域とデータ領域に分かれている。そして、前者には i-ノードと呼ばれるレコードが順番に詰め合わさって入っていて、その一つ一つがファイルまたはディレクトリに対応している。ファイルもディレクトリも持ち主、モードなどの属性を持っているが、これらはこの i-ノードに格納されている。

一方、ファイルの中身は? それはデータ領域にある、データブロックに格納されていて、i-ノードにこのファイルのデータブロックはどれとどれ、という情報が入っている。このように分けてあるのは、ファイルの本体というの

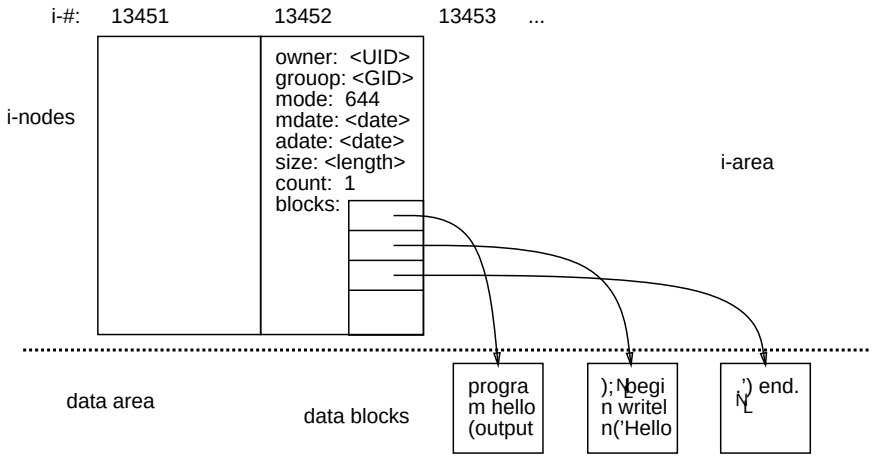

図 3.10: i-ノード、i-領域とデータ領域

は大きさが伸びたり縮んだりするので、分けてある方が管理しやすいからである。

さて、話を戻して、ファイルの名前が他の属性と同じように i-node の中に格納されているとすると、いろいろ困ったことが起こる。例えば長さがひどく長くてもよいのでその最大の長さぶんの場所を取ると大変である。また、ln で名前が 2 つつけられるのはどうしようか? そして何よりも、ある名前のファイルを探す時に、i-領域を頭から順番に見ていくわけには行かないだろう。

というわけで、ファイルの名前というのはファイルの中には入っていない。ではどこに入っているか? (もうお分かりですね?) ディレクトリである。図 3.11 左に示すように、ディレクトリは実はファイル名と i-番号の対応を記した表にすぎない。が、これでちゃんと右側に対応する情報が記されているわけである。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だろうか。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である (普段はあまり意識する必要はないが)。また、「.」とか「..」は自分自身、および親へのリンクだということになる。

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る。図 3.12 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、という意味である。また、rm はリンクを切る指令であるが、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

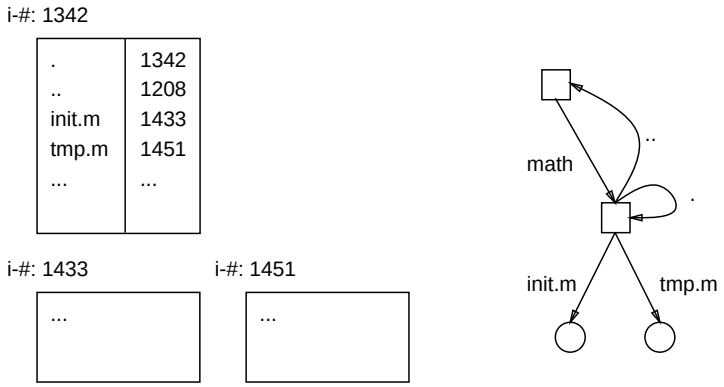

図 3.11: ディレクトリの中身

れ、その領域は回収される。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る。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る。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るが、これは ln+rm ではできない。というのは、Unix ではディレクトリに複数名前をつけることは許されないようになっているからである。

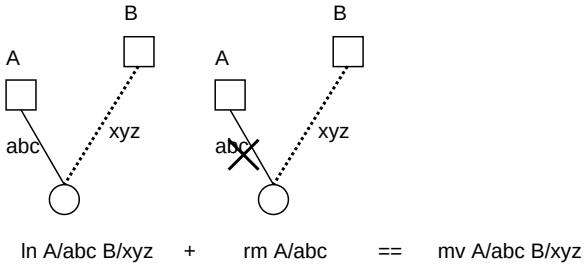

図 3.12: ln と rm と mv の関係

3.8 ファイルシステムのその他の話題

3.8.1 シンボリックリンク

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ない。さらに、普通のファイルでもディスクが違う場合には張ることができない。こういう制限はもっともなことではあるが (なぜもっともかな?) 不便でもある。

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るものが多い。

```
ln もとのパス名 新しいパス名       -- 普通のリンクを張る
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
```

のように、どちらも同じ ln 指令を使い、意味も類似している。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」わけである

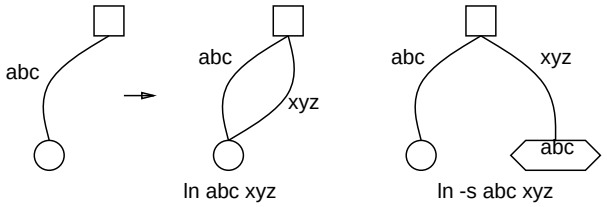

図 3.13: シンボリックリンクの概念

シンボリックリンクは上述のようにディレクトリを指すことができ、またディスクをまたがってもいいのでファイルの構造を整理するのに便利である。

3.8.2　ディレクトリ単位の操作

上で述べたように、mv を使えばディレクトリごと場所を移動できるのでファイルの整理に便利である。またそれ以外にも

```
mv ファイル... ディレクトリ   -- 複数のファイルをいちどに移動
cp ファイル... ディレクトリ   -- 複数のファイルをいちどにコピー
```

という使い方もできる。これらの場合は名前そのものはもとのまま、ということになる。またディレクトリの木構造をそのままにそっくりコピーしたり、木構造をそっくり削除するには

```
cp -r ディレクトリ 行き先
rm -r ディレクトリ
```

が使える。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。

3.8.3　領域管理、探索

自分のファイルが増えてくるとその管理も大変である。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

`du` ディレクトリ　　-- その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 -- 同様だが、個々のファイル名と大きさも表示

によるとよい。

大きさだけでなく、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find である。これには色々なオプションがあるがいくつかの例を挙げる:

`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 nc -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。

3.9 演習

3-1. ファイルのモードをいろいろ変更し、ls で正しく変更されているか確認せよ。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よ。また、「誰にでも読めるが自分には読めない」といった変な設定が意味を持つかどうか試してみよ。

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よ。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つか? その中の個別のファイルに対する保護とはどう違うか?

3-3. 隣に座った人と協力して、自分のホームディレクトリの保護モードを「誰にでも読める・実行できる」や「誰にでも書ける」にした上で、隣の人が自分のディレクトリやファイルを読めるか、またその人が自分のホームディレクトリにファイルを作れるかなど確認してみよ。

3-4. `t03.c`〜`t06.c` を打ち込んでコンパイルし、動かしてみよ。また BUFSIZE を変化させてみて、どれくらいの値がよさそうか検討せよ。

3-5. 例題を参考にして「ちょうど 10,000 バイトのファイルを作るプログラム」を書いて動かし、できたファイルの大きさが 10,000 バイトであることを確認せよ。