第4章　UNIX

この章では UNIX オペレーティングシステムの概略, 利用法などについて解説する. Section 4.1 は UNIX の歴史・概略について述べてあるが, この部分は UNIX オペレーティングシステムに対してのイントロダクションと理解すればよい. それに続く section は, 実際に UNIX を利用した後に雰囲気が理解できれば十分である. また, UNIX を利用した後に Section 4.1 を読み直すことで UNIX に対する理解がどれだけ進んだかを何度も確認することができる.

4.1　UNIX とは

UNIX はワークステーションに搭載されている, AT&T に起源をもつ OS の一群を指す名称である. UNIX は 1969 年に AT&T のベル研究所で Dennis Ritchie などによって, DEC PDP-7 にはじめて搭載された. その後, 1973 年には Ritchie たちは, UNIX の基本的な部分を C 言語によって記述し, それまでは OS はアセンブラで書かれるという常識をうち破り, 現在の UNIX の基礎を築いた. このように, UNIX は C 言語によって記述されているので, 広範囲のシステムに移植可能であり, それによって, 多くのシステムの上で実際に UNIX が動いているのである. UNIX は長い歴史の間に各種のメーカにより作られたものも多く,

- Sun Microsystems による Solaris.
- DEC による OSF1, 今日では, Digital UNIX Tru 64 となっている[1].
- Silicon Graphics による IRIX.
- IBM による AIX.
- NeXT による NeXTstep と Apple による MacOS X （正しくは Rhapsody (MacOS X Server), Darwin (MacOS X) と呼ぶべき）

など多くのベンダー UNIX が存在する一方で, Linux, FreeBSD, NetBSD などのフリー UNIX 等が存在している. これらの UNIX は, それぞれのソース・コードの由来等により, BSD 系列のものと, AT&T 系列 (System V) に分けることも可能であるが, 今日の BSD 4.3 等では, SVR4 (System V Release 4.0) のコードも含まれていて, 厳密にそれらの系列をたどることは難しい.

UNIX の基本的な特徴の一つは, **タイムシェアリング** (**Time Sharing**: **時分割**) による, 複数のユーザの CPU の利用を核とした, マルチタスクの OS であるということである. また, 外部記憶装置, 入出力装置などをデバイスという概念によって統一したことによって, それらの装置の扱いが容易になったばかりではなく, デバイスの追加という単純な操作で, 新しい装置を追加することができるという可搬性も特徴になっている.

実際に UNIX を利用するためには, キーボードからの入力を UNIX のカーネル（UNIX の最も基本的な部分）に伝える方法が必要となる. UNIX のカーネルは複数のユーザそれぞれに対して, 端末と呼ばれるデバイスを割り当てる. 端末からの入力, 端末への出力と UNIX カーネルとの仲介をとっているプログラムがシェルである. シェルとは, キーボードからの入力の各行を解釈して, コマンドとして UNIX のカーネルに渡したり, 入力されたコマンドの出力をディスプレイに出力したりする役割を持っている. UNIX では標準的には sh (Boune shell) と呼ばれるものと, csh と呼ばれるものの２つが用意されている[2]. それぞれのシェルは対話モードと非対話モードの２つのモードがあり, 通常利用するのは対話モードである. どちらのモードでも, プログラミング可能である. 非対話モードに関しては, シェルスクリプトに言及することができれば, そこで説明する. 対話モードでは, シェルはプロンプトと呼ばれる記号を出力し, プロンプトがある場合に限ってコマンドを入力することができる.

[1] DEC 社は Compaq 社に吸収合併されたため, Compaq Digital UNIX Tru 64 となった. さらに, Compaq 社は Hewlette-Packard 社に吸収された.

4.2　ログインとパスワード

4.2.1　ログインとパスワードの意味

UNIX ワークステーションはマルチ・ユーザの OS であるため, UNIX を利用するためには, **ユーザ名** (**user name**) と**パスワード** (**password**) の組による認証[3]を行わなくてはならない. このことを**ログイン** (**login**) と呼ぶ. ログインは, ワークステーションの利用が許可された複数のユーザを識別する手続きであり, ログインを要求しているユーザの正当性を確認する方法がパスワードの入力である.

なお, UNIX ワークステーションの利用を終了するときには, **ログアウト** (**logout**) と呼ばれる操作により, ログインする前の状態に戻す必要がある.

通常 UNIX のユーザ名は８文字以下の英数字および記号からなり, パスワードは最大識別文字数が通常８文字である[4]. ユーザ名は大文字と小文字を区別しないが, 大文字でユーザ名を入力すると, 小文字が利用できない端末と見なされる. **パスワードは, 英数字および記号からなり, 英字の大文字と小文字が区別される.**

UNIX では, ユーザ名やパスワードなどの**ユーザ情報** (**user information**) は, ユーザ認証システム内のデータとして保存されている. ユーザ認証システムでは, 通常はユーザのパスワードは暗号化された形で保存されていて, 入力されたパスワードを暗号化したものと, 認証システム内の暗号化パスワードが一致しているときに, 正しいパスワードが入力されたものと判断される.

4.2.2　ユーザ認証システム

UNIX システムにおける, ユーザの正当性を認証する一連の仕組みを**ユーザ認証システム** (**user authentication system**) と呼ぶ. もし, UNIX ワークステーションが単独で動作している場合には, ユーザ認証システムのユーザ情報は, そのワークステーション内にファイルとして保存され, 特にユーザのパスワードは暗号化された形で保存されている. この場合の認証システムは, アプリケーションに入力されたユーザＩＤとパスワードを, 保存されているデータと照合することによりユーザ認証を行う. 通常, ユーザ情報は `/etc/passwd` というファイルに保存される. 近年のシステムでは, `/etc/passwd` とは別に暗号化パスワードを保存する `/etc/shadow` というファイルを用いているものが多い. （これを**シャドウパスワード** (**shadow password**) と呼ぶ.）

しかし, ある程度の規模の複数台のホストからなる UNIX システムは単独の機器で動くことは少なく, コンピュータネットワークを介して, 複数の UNIX ワークステーションが連携して動作していることが多い.

[2] 今日の UNIX では, sh を拡張した bash, csh を拡張した tcsh 等も標準的に搭載されることが多く, さらに, ksh (Korn shell), zsh などというシェルも搭載されていることが多い.

[3] Authorization, Authentication, Accounting と呼ばれる操作であり, 接続元ホストやユーザ名を Authorization し, ユーザ名とパスワードによって Authentication し, そのユーザに対する Accounting を開始するという手続きが行われる.

[4] パスワードの最大識別文字数や, 最小必要文字数はシステムの構成によって変更することも可能である.

このような場合, 各ホスト（各 UNIX ワークステーション）にそれぞれ独立の認証システムを導入すると, 認証データの間に矛盾[5]を生じたり, システムの管理に手間がかかったりする. そのため, ネットワークを介して, 認証データを一元化する認証システムが広く導入されている. このようなネットワークを介した認証システムの代表例としては, Sun Microsystems 社が設計した **NIS** (**Network Information System**) と呼ばれるシステムが, 古くから広く用いられている. NIS は UNIX ベースのシステムでは標準的に採用され, 現在でも幅広く用いられている認証システムである. また, NIS は単に「ユーザ情報」だけを共有するシステムではなく, UNIX ホスト間で共有すべき多くのシステム情報を共有するためのシステムとして設計されている.

近年, ユーザ認証システムにより多くの情報を持たせようとする試みや, UNIX に限らない他のシステム（Windows など）との情報の共有を行おうとする試み, さらには, 単独の認証システムではなく, 複数のシステムの間でそれぞれがもつ情報の部分的な共有といった, 複雑な情報共有システムの構築の試みが広まっている. さらに, 公開されたネットワークを用いて情報の共有を行おうとするためのセキュリティの確保という課題も浮上している.

このような情報共有システムを NIS を用いて実現することはかなり困難が付きまとい, そのシステム構築には高い技術レベルと十分な管理能力が必要となる. そのため, セキュリティ技術との連携が容易で, 情報の複雑な共有機構をもち, UNIX に限らない他のシステムでも利用可能な認証システムとして, **ディレクトリサービス** (**directory service**) と呼ばれるシステムを認証システムに用いることが多くなっている. ディレクトリサービスの代表例としては, 古くは NeXTstep で採用され, 近年 MacOS X でも採用された **NetInfo**, 現在多くのシステムで利用可能となっている **LDAP** (**Lightweight Directory Access Protocol**) がある.

また, ユーザ認証システムは, UNIX などのホストの利用のための認証だけではなく, 「インターネットサービスプロバイダ」や「無線ＬＡＮシステム」の利用という, ネットワークの利用という場面でも必要とされるシステムである. このような認証システムでは, ユーザＩＤとパスワードの組を受信するシステムが UNIX などの「ホスト」ではない場合が多いため, 受信するシステム上で直接認証システムを動作させることが困難なため, 「外部認証」と呼ばれるシステム, すなわち, 受信したユーザＩＤとパスワードの組を認証サーバに送信し, 認証結果だけを得るという仕組みも実現されている[6].

4.2.3 パスワードの変更

パスワードは定期的に変更した方が良いと言われている. しかし, 余りに頻繁に変更して, パスワードを忘れたり, それを防ぐためにパスワードのメモを残すこととなっては本末転倒となる.

古典的な単独の認証システムでは, パスワードを変更するためのコマンドは, `passwd` コマンドを用いる.

【利用法】 `passwd`

このコマンドを入力すると, はじめに現在のパスワードの入力を求められ,

```
Old Password:
New Password:
Retype New Password:
```

その後, 新規パスワードの入力を２度求められる. ２度の新規パスワードの入力結果が一致しているときに限り, 新規パスワードへの変更が行われる.

[5] あるホストではパスワードを変更したのに, 他のホストにその変更が伝達されないということなど.
[6] このような認証システムの代表例として **radius** と呼ばれる認証システムが有名である.

NIS, NIS+, LDAP などの認証システムを使っているワークステーション群の場合には, `yppasswd` や, `passwd -y`, `passwd -r nis`, `passwd -r nisplus`, `passwd -r ldap` などという, 認証システムに対応したパスワード変更コマンドを利用しなければならない. これらの認証システムのもとでのパスワード変更方法にについては, ＯＳに依存している部分も多いため, オンラインマニュアルや各サイトのガイドを調べる必要がある.

4.3 UNIX のファイルシステム

ここでは, ファイルとディレクトリの概念を見ていこう.

4.3.1 ファイルとは

ファイル (**file**) とはディスクに保存した名前の付けられたデータの集合の単位のことである. UNIX は一つ一つのファイルをその名前を利用してアクセスする.

4.3.1.1 ファイル名

ファイルの名前には, `/` を除く全ての英数字・記号・漢字などの日本語などが利用できる. しかし, 記号 (`(`, `)`, `\`, `*`, `?`, `&`, 空白, 及びファイル名の先頭の `-`) や日本語を利用することは余り良いことではない[7]. また, 通常の UNIX のファイルシステムでは, ファイルの名前の長さは, １文字以上２５５文字以下であるが, 極端に長い名前をつけるのも関心しない.

コンピュータを使う上での常識として, ファイルの名前は, それを見てどのようなファイルであるかすぐにわかるような短い名前をつけるのが良い. また, `XXXX.YYY` というように, `.` で区切って `YYY` の部分にそれが何のファイルであるかを示すような文字列（**拡張子** (**extension**) と呼ぶ）をつけることが多い. 例えば, 以下のような付け方をする.

- `XXXX.p` : PASCAL の Program のファイル.
- `XXXX.c` : C の Program のファイル.
- `XXXX.tex` : TEX のファイル.

UNIX のファイルシステムにおいては, 拡張子は基本的には意味を持たない. OS がそのファイルがどのようなファイルであるか（実行形式かどうか等）を判断するためには, **マジックナンバー** (**magic number**) と呼ばれる数値を利用する. これは, ファイルの先頭数バイトが特徴的なデータをなすことが多いため, そのデータベースと照合することによって, ファイルがどのようなものであるかを知ることが出来る.

しかし, UNIX 上のアプリケーションには, 拡張子の形式によって処理方法を変えたり, 特定の拡張子でなければ処理を行わない等という仕様になっているものも多い. （例えば, `make` コマンド, 各種言語処理系など.）そのため, 適切な拡張子をファイル名に与えることは UNIX においても重要である[8].

[7] ファイル名に記号を用いると, シェルで利用される特別な意味を持つ記号との混乱が生じる. また, 日本語を用いると, 日本語文字コードが異っていても一見同じファイル名に見えたりすることが混乱の元となる. さらに, UNIX のコマンド群では, 日本語の解釈を正しく行わないものもある.
[8] MacOS は拡張子に相当する概念を持たない. MacOS ではそれぞれのファイルに対して「タイプ」と「クリエータ」と呼ばれる属性が与えられている. MacOS はこの２つの属性によって, ファイルが実行形式かどうか, そのファイルを作成したアプリケーションが何かを判断する.
一方, Windows では, 拡張子とファイル形式の間の対応が決まっていて, 拡張子を変えると適切な実行や, アプリケーションの呼び出しに障害が発生する. 正直に言えば, 「信じられない低レベルの発想」である.
MacOS X は, コマンドレベルでは UNIX (BSD) として動作するので, 通常の UNIX と同じと考えて良いのだが, ファインダ

4.3.1.2 ファイルのオープンとクローズ

システムがファイルからデータを読み出すとき, システムがデータをファイルに書き出すときには, ファイルを**開く** (**open**) という操作が行われる. ファイルを開くとは, プロセスのファイル記述子と呼ばれる変数にファイルのエントリを対応させる操作であり, 書き出しモードでファイルを開いた場合には, ファイルシステムにそのファイルのデータを格納する領域が確保される. ファイルからのデータの読み出しや, データのファイルへの書き出しが終了した時点では, 必ずファイルを**閉じる** (**close**) という操作が行われる. これは, プロセスのファイル記述子を開放する操作であり, 書き出しモードでファイルを開いている場合には, ファイルシステムに対して, データ領域に終了フラグをたてることを要求する. 特に, 複数のプロセスがファイルを開いている場合には, 最初に書き出しモードでファイルを開いたプロセスにより, ファイルに対して排他制御ロックがかかっていることがあり, この場合は他のプロセスは書き出しモードではファイルを開くことができない[9].

4.3.2 ディレクトリとは

ディレクトリ (**directory**) とは, ファイルを格納する単位になるもので, 階層構造を持っている[10].

4.3.2.1 ディレクトリの構造

UNIX のファイルシステムでは**ルート・ディレクトリ** (**root directory**) と呼ばれる, 階層構造の一番上にあるディレクトリから木構造をなしている.

例えば, ルートディレクトリの下には `/bin`, `/usr`, `/dev` などのディレクトリがあり, その下にもいくつものディレクトリがあるという構造である. このようなものを**サブディレクトリ** (**subdirectory**) と呼び, サブディレクトリから見てすぐ上のディレクトリを**親ディレクトリ** (**parent directory**) と呼ぶ. ここで, ルートディレクトリを `/` という文字で表現した. ルートディレクトリには親ディレクトリは存在しない.

ルートディレクトリを除く全てのディレクトリには, 親ディレクトリと自分自身を示す特別な**ディレクトリエントリ** (**directory entry**) が存在する. (ディレクトリエントリとは, ディレクトリを指し示す[11]ファイルである. UNIX ではディレクトリもディレクトリエントリを利用してファイルと見做していることに注意.) 各ディレクトリでは, それ自身を指し示すディレクトリエントリは `.` で, 親ディレクトリを指し示すディレクトリエントリは `..` で表現されている.

シェルは (それが対話モードであっても, 非対話モードであっても) いつでも現在いるディレクトリを知ることができる. それを**カレントディレクトリ** (**current directory**) と呼ぶ. また, ログイン時にカレントディレクトリとなるディレクトリを**ホームディレクトリ** (**home directory**) と呼び, csh では `~/` で表すこともできる.

なお, 異なるディレクトリにある同じ名前のファイルは, 異なるファイルとして見做される. これは後に述べるパスを使って理解すればよい.

(Aqua) のレベルになると, 旧来の MacOS との互換性を残すために, 相変わらずファイルの「タイプ」と「クリエータ」を利用している. そのため, MacOS X のファインダでは, 拡張子によるファイルとアプリケーションの対応付けと同時に, 「クリエータ」による対応付けが併用されている.

[9] UCB Mail コマンドは, 実行時にメール・スプール・ファイルに対して, BSD タイプの排他制御ロックを行う. したがって, UCB Mail コマンド実行時に他のプロセスによってスプール・ファイルを書き出しモードで開くことは出来ない. しかし, BSD タイプの排他制御ロックは NFS (Network File Sharing System) を経由した場合には無効となるので, NFS により複数のホストによってスプールが共有されている場合には, UCB Mail コマンドを複数のホストから実行すると, スプール・ファイルが破壊される場合がありうる.

[10] MacOS や Windows で言うところの**フォルダ** (**folder**) と同義語であると考えて良い. Windows では, OS のレベルではフォルダはディレクトリそのものであり, MacOS でもほぼ同等の概念である. MacOS X でのフォルダは完全にディレクトリそのものである.

[11] このようなものをポインタと呼ぶ.

4.3.2.2 ホームディレクトリ

UNIX では, 一般のユーザは, 各自のホームディレクトリ以下にしか書き込み権限を持たない.[12] 逆に言えば, ホームディレクトリ以下は各ユーザが排他的に利用できる領域であり, (ファイル容量制限を越えない限り) ホームディレクトリにはどのようなファイルを持つことも自由である.

もし, ホームディレクトリ内のデータを他人にみられたくなければ, 後に示す許可モードや `umask` を使って, ホームディレクトリのデータの許可属性を変更する必要がある.

4.3.2.3 ファイルのパス

それでは, 色々なディレクトリにあるファイルをどのように指定できるかを見てみよう.

まず, カレントディレクトリにある `A` という名前のファイルは, 単純に `A` で指定できる. しかし, それ以外にも指定の方法がある. カレントディレクトリがルートディレクトリの下の `home` というディレクトリのまた下の `you` というディレクトリである場合, `A` は

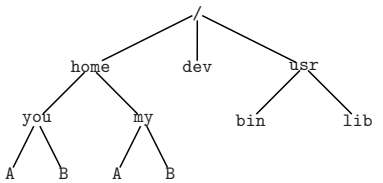
```
/home/you/A
```

と表現できる. これをファイルの**絶対パス** (**absolute path**) と呼ぶ.

一方, 次のような指定もできる.

```
./A
```

これは, `.` がカレントディレクトリを示しているので, カレントディレクトリの下の `A` というファイルを指定したことになる. これをファイルの (カレントディレクトリからの) **相対パス** (**relative path**) と呼ぶ. このように, `/` という記号はディレクトリの区切りを表している.



Exercise 4.3.1 カレントディレクトリがルートディレクトリの下の `home` というディレクトリのまた下の `you` というディレクトリである場合, `home` の下の `my` というディレクトリにある `X` というファイルを絶対パスと相対パスで表現せよ.

4.3.3 ファイルのパーミッション

UNIX はマルチユーザの OS であるため, ディレクトリなどを含む各ファイルには**パーミッション** (**permission**) と呼ばれる, 許可属性が付けれらている. 各ファイルには**所有者** (**owner**), **所有グループ** という属性があり, 各ファイルは `user`, `group`, `other` ごとに, `read`, `write`, `execute` という許可属性を持つ.

UNIX ではすべてのユーザは 1 つ以上の**グループ** (**group**) に属しているので, `group` に対する属性とは, ファイルの所有グループに属するユーザに対する許可属性であり, `other` に対する属性とは, ファイルの所有グループに属さないユーザに対する許可属性である.

[12] 実際には, このほかに, `/tmp` で表されるものを代表とする, 一時的な記憶領域を利用することも可能ではあるが, それらはシステムによって定期的に消去されたりする.

4.3.3.1 パーミッションの表現方法

パーミッションは, user, group, other ごとに, 8進数または, 記号で表現される. 8進数表現では, read 属性のある場合には 4, write 属性のある場合には 2, execute 属性のある場合には 1 が加えられ, 0 から 7 の数値で表現される. これらを user, group, other の順に並べた数値がパーミッションの数値表現である.

記号の場合は, rwx, r--, rw- などのように, read, write, execute の順に, 属性がある場合にはその頭文字を, 無い場合には - を並べ, この記号を user, group, other の順に並べた数値がパーミッションの記号表現である.

4.3.3.2 パーミッションの意味

read, write 属性に関しては, その意味を理解するのは易しい. それぞれのユーザに対する許可属性がある場合にのみ, ユーザはその操作を実行できる.

execute 属性に関しては, ディレクトリの場合と, 通常のファイルの場合で意味が異なる. 通常のファイルの場合に execute 属性があれば, そのファイルを実行することが可能であることを示す[13] ディレクトリの場合に execute 属性があれば, そのディレクトリ内部のファイル一覧を得ることが出来ることを示す. したがって, 実行形式ファイルに execute 属性がついていない場合には, そのファイルを実行することは出来ない.

Example 4.3.2 以下ではいくつかの許可属性に関して考察をしてみよう.

1. 所有者のみが読み書きすることができ, 他のユーザが一切読み書きすることができない許可属性は, 8進表現で 600 である.
2. 所有者のみが読み書きすることができ, 他のユーザは読むことだけができる許可属性は, 8進表現で 644 である.
3. 所有者のみが書き込み読み出しができて, 他のユーザが一切読み出しも書き込みもできないディレクトリの許可属性は, 8進表現で 700 である.
4. 所有者のみが書き込み読み出しができて, 他のユーザが一切読み出しのみができるディレクトリの許可属性は, 8進表現で 755 である.
5. 所有者のみが書き込み読み出し実行ができて, 他のユーザが一切読み出しも書き込み実行ができない実行形式のファイルの許可属性は, 8進表現で 700 である.
6. 所有者のみが書き込み読み出し実行ができて, 他のユーザが一切読み出しと実行のみができる実行形式のファイルの許可属性は, 8進表現で 755 である.
7. 実行形式のファイルが 511 という許可属性値を持つときには, 所有者はそのファイルの中身を読み出すことと実行することができる. しかし, 他のユーザはファイルの中身を見ることができない. それにも関わらず, その実行形式のファイルを実行することが可能である.
8. ディレクトリが 711 または 511 という許可属性値を持つときは, 所有者以外のユーザはそのディレクトリのファイル一覧を見ることができないが, そのディレクトリの中にある個々のファイルに対する読み出し属性がある場合には, 個々のファイルは読むことができる. また, そのディレクトリの中にある個々のファイルに対する実行属性がある場合には, 個々のファイルは実行することができる.

[13] より正しくは, UNIX の実行可能形式のファイルであっても, execute 属性がついてない限り, それを実行できないことを意味する.

9. ディレクトリが 744 または 544 という許可属性値を持つときは, 所有者以外のユーザもそのディレクトリのファイル一覧を（ある意味で）見ることができる. しかし, そのディレクトリの中にある個々のファイルに対して読み出し属性があっても, 個々のファイルを読むことができない.
10. ディレクトリが 733 という許可属性値を持つときは, 所有者以外のユーザは, そのディレクトリにファイルを作ることが可能であるが, 作ったファイルを読み出すことはできない.

Example 4.3.3 あるディレクトリに write 属性があり, そのディレクトリ内のファイルに対して write 属性が無い場合, そのファイルに対してどのような操作が可能であるかを考えてみよう.

具体的には, ls コマンドでの出力が

```
drwxrwxrwx   2 root     other        ./
drwxrwxrwt  12 sys      sys          ../
-rw-r--r--   1 root     other        a
```

となっているときに, 一般ユーザがファイル a に対して行える操作を考えてみる.

- ファイル a の中身を変更することは不可能. (EDIT)
- ファイル a の名前を変更することは可能. (MOVE)
- ファイル a を消去することは可能. (REMOVE)
- ファイル a をこのディレクトリ内でコピーすることは可能. (COPY)

これらの操作のうち, MOVE, REMOVE, COPY はファイルそのものに対する操作ではなく, ディレクトリエントリ（ディレクトリのファイルアロケーションテーブル）に対する操作である. ディレクトリに対する操作は, ディレクトリに対する write 属性によって許可されているので, これらの操作が可能になる. しかし, EDIT だけは, ファイルそのものに対する操作であるため, ファイルに対する write 属性がなければ行うことができない[14].

当然, このディレクトリに新規ファイルを作成することも可能になる.

4.3.3.3 パーミッションの変更

新規に作成したファイルの許可属性は, シェル変数 umask の値に, ファイルの場合には 666 を, ディレクトリの場合には 777 をマスクした（ビットごとに NAND をとった）値が用いられる.

Example 4.3.4 umask の値が 077 の場合, 新規にファイルを作成すると, そのファイルの許可モードは 600 となる.

既存のファイルの許可属性は, ファイルの所有者以外は変更できない. また, ファイルの所有者属性は root ユーザ[15]（システム管理者）以外は変更できない.

ファイルのパーミッションを変更するためには chmod コマンドを用いる. また, ファイルの所有者を変更するためには chown コマンドを用いる.

[14] ファイルの情報（許可属性, タンムスタンプ, ファイルサイズなど）を得るためには, ディレクトリに対する read 属性が必要となる.

[15] UNIX では, 全面的な権限を持った root と呼ばれるユーザが存在する. 「特権ユーザ」とも呼ばれ, いかなる許可属性がついていようとも, root はすべてのファイルを変更する権限を持つ.

4.3.4 リンク

UNIX のファイルには, **リンク** (**link**) と呼ばれる概念がある. これは, 一つの（実体を持った）ファイルを複数のファイル名（正確にはパス名）で表現することである. 例えば, `/home/naito/X` というパス名を持ったファイルと `/home/naito/Y/a` というパス名を持ったファイルがリンクされているということは, どちらのパスを利用してファイルを参照しても, 同じ実体を参照することになる. これを**ハード・リンク** (**hard link**) とも呼び, リンクを実現するには, ファイルシステムのファイル・アロケーション・テーブル上にある, パス名を表すエントリに同一のセクタを指示することによって実現するため, 同じファイルシステム内にあるファイル同士でしかリンクすることは出来ない.

ファイルを削除する, 移動すると言った操作は, ファイル・アロケーション・テーブル内のエントリの変更によって実現されるため, リンクのある（正確にはリンク数が 2 以上の）ファイルに対してこれらの操作を行っても, 他のリンク・ファイルには影響は及ばない.

一方, 良く似た概念に**シンボリック・リンク** (**symbolic link**) がある. シンボリック・リンク・ファイルの中身が, 他のファイルのパス名になっているもので, シンボリック・リンク・ファイルを参照する場合には, 実際には, そのリンクが指し示す先のファイルを参照することとなる. MacOS で実現されている「エリアス」に似た概念だが, 実現方法は全く異なり, 特にリンク先のファイルを移動した場合には, MacOS のエリアスでは自動的にリンク先が変更される[16]が, シンボリック・リンクでは, もとのリンクファイルの内容は変更されない.

```
12056 drwx------   9 naito  math      2560 Mar  7 16:15 .
12000 drwx------  11 naito  math       512 Feb 25 17:36 ..
18963 drwx------   9 naito  math       512 Jul  3  2000 example
13566 -r--r--r--   2 naito  math     14081 Mar  6 08:33 unix-command.tex
13566 -rw-------   2 naito  math     18356 Mar  7 16:04 unix0.tex
14876 -rw-------   1 naito  math     18356 Mar  7 16:04 unix1.tex
18098 lrwxrwxrwx   1 naito  math         9 Mar  7 16:04 unix2.tex -> unix1.tex
```

`ls -lgi` コマンドでファイルの一覧をとった例である. 先頭の数値は **i-node** と呼ばれる, ファイル・アロケーション・テーブル内でファイルを一意に識別する数値である. 次の１０文字が許可モードを表す記号, リンク数, 所有者, 所有グループ, ファイルサイズ, 最終変更日時, ファイル名の順で表示されている. `.`, `..`, `example` はディレクトリであり, 許可モードの欄の先頭に `d` がついている. `unix0.tex`, `unix-command.tex` はリンクされている. このことは, リンク数が 2 となっていること, i-node 番号が一致していることからわかる.
`unix2.tex` は `unix1.tex` へのシンボリック・リンクになっている. ファイルサイズが `unix1.tex` という文字列のバイト数である “9” になっていることに注意. また, シンボリック・リンクの許可・所有者属性は, リンク先のファイルの属性が使われるため, ここに表示されている属性値に意味はない.

4.3.5 ファイルシステム

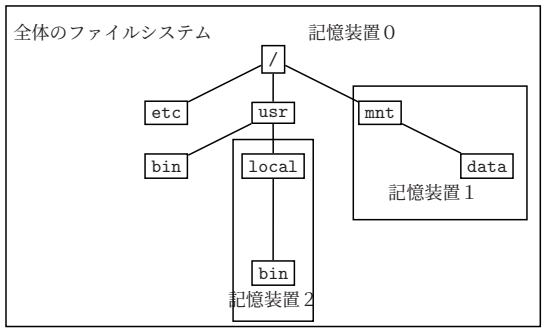
Windows などのパーソナルコンピュータでは, 内蔵されたハードディスクには `C:` という「ドライブ名」がつけられ, フロッピーディスクには `A:` という「ドライブ名」がつけられている. また, MacOS では, 内蔵されたハードディスクや挿入されたフロッピーディスクは, デスクトップ上にアイコンとして表示されている. したがって, これら Windows や MacOS では, その機器に接続されている記憶装置（記憶メディア）には, それぞれ, ドライブ名や機器のアイコンを通じてアクセスすることが可能である.

それでは, UNIX ワークステーションでは, 機器に接続されているそれぞれの記憶装置はどのように認識されているのだろうか. UNIX ワークステーションでも, 機器に接続されている記憶装置は, 必ずしも一つ

[16]エリアスはパス名ではなく, ファイルの「カタログBツリー」テーブル内の ID を参照している.

ではなく, 複数のハードディスクが使われていたり, 場合によってはフロッピーディスクや CD-ROM が挿入されることがある. UNIX では, ユーザからみたとき, このような各記憶装置全体を単一の**ファイルシステム** (**file system**) と見なす.

具体的には, UNIX システムの起動システムに利用される記憶メディア（実際には, ほとんどの場合ハードディスク）の**ディレクトリツリー** (**directory tree**), すなわち, その記憶メディアのデータがなすディレクトリ構造をルートディレクトリと見なすことから始まる. その後に, それ以外の記憶メディアのディレクトリツリーを, ルートディレクトリ以下のどこかのディレクトリに**マウント** (**mount**) することにより, 複数の記憶装置のデータ全体を単一のファイルシステムと見なす.



「記憶装置０」は「起動ディスク」である. 起動ディスクのディレクトリツリー内の `/mnt` には, 「記憶装置１」のディレクトリツリーがマウントされ, 起動ディスクのディレクトリツリー内の `/usr/local` には, 「記憶装置２」のディレクトリツリーがマウントされて, 全体が一つのファイルシステムに見えている.

さて, 一つのファイルシステム内で, 複数の記憶装置が利用されていることが, ユーザがファイルを扱う上で, どのように関係しているかを考えてみよう.

あるファイルの名前をつけ変えること（ファイルを移動すること）を考えてみよう. （そのためには, 後に解説する `mv` コマンドを利用する.）

元のファイルと移動先のパスが同一の記憶装置の中にある場合 この時, ファイルの移動とは, 単に**ディレクトリエントリ** (**directory entry**) 内の名前のつけ変えに過ぎない. すなわち, 記憶装置の中で, そのデータがどこに格納されているかという表（これが「ディレクトリエントリ」である. UNIX では **i-node** と呼ぶ.）で, ファイルの名前を書いてある部分を書き換えているに過ぎない. したがって, この場合には `mv` コマンドは一瞬にして終了する.

元のファイルと移動先のパスが異なった記憶装置の中にある場合 この時, ファイルの移動は, ディレクトリエントリの書き換えではなく, 実際にデータを移動先の記憶装置に書き込み, 移動先の記憶装置のディレクトリエントリをつくった上で, 移動元の記憶装置のディレクトリエントリを消去する. したがって, この場合には `mv` コマンドは, 実際にはコピーと消去の組み合わせとなる.

UNIX では, 各個人のホームディレクトリは, 単一の記憶装置内に存在するため, 個人のホームディレクトリ内のファイルの移動では, このようなめんどうなことは生じない.

実際には, UNIX のファイルシステムは, 各ホストに接続された記憶装置だけではなく, ネットワークを介して, 他のホストに接続された記憶装置をマウントすることも可能である. （これを **NFS** (**Network File System**) と呼ぶ.）この NFS を利用することにより, 多くの UNIX ワークステーションの間でデータを共有し, 機器が異なってもユーザに対して同一の環境を提供することが可能になる.

4.3.6 クオータ

UNIX では各ユーザの所有するファイルの全容量をある一定値以下に制限することができる．これを**クオータ** (**quota**) という[17]．クオータはファイルの全容量だけではなく i-mode 数（おおざっぱに言えばファイルの総数）にも制限を置くことが可能である．

通常クオータは **ハードリミット** (**hard limit**) と**ソフトリミット** (**soft limit**) の２種類の上限値があり，以下のように動作する．（ここでは，ハードリミット値を H，ソフトリミット値を S とする．前提条件として，$S \leq H$ である．）

1. ファイル全容量 x が $S \leq x \leq H$ を満たしている継続期間が，システムによって定められた時間内であれば，ファイルを新たに作成することができる．しかし，この継続期間がその時間を越えると，ファイルを新たに作成することができなくなる．
2. ファイル全容量 x が $x \geq H$ になった場合には，ファイルを新たに作成することができなくなる．

この継続期間はシステムによって異るが，１週間程度に定められていることが多い．

なお，クオータは正確にはファイルシステムごとに決まる概念で，クオータの計算もファイルシステムごとに計算される．

4.4 UNIX のシェル

ここでは，UNIX のシェルの対話形式による利用法を簡単に解説する．対話形式でのシェルは，キーボードからのコマンドの実行命令を解釈して，UNIX カーネルに伝達する役割を果たす．

以下では，通常コマンドインタープリタとして利用されることの多い csh に関して解説を行う．

4.4.1 シェルからのコマンドの実行

シェルからコマンド実行するには，**プロンプト** (**prompt**) と呼ばれる記号の後に，実行したいコマンドとその引数を書き，↵を押せばよい．

Example 4.4.1

```
%␣cp␣a␣b
```

ここで，% がシェルのプロンプトであり，その後に cp というコマンドを，２つの引数 a, b をつけて入力した．

このように，コマンドや引数は「空白文字」で区切られる．もし「空白文字」を含むファイル名などを引数にしたい場合には，\ を用いて「空白文字」をエスケープすればよい．

Example 4.4.2 もし，ファイル名が cmd␣1 という名前であったときには，cmd\␣1 とすれば「空白文字」をエスケープしたことになる．

4.4.2 シェルによるワイルドカードの展開

シェルからコマンドを入力する際には，ファイル名を指定しなくてはならないことが多い．その際に，ファイルの名前のパターンを指定して，複数のファイルにマッチさせることができる．そのようなファイルの指

[17] 正確には**ファイルクオータ**と呼ぶ．

定の方法を**ワイルドカード** (**wild card**) によるファイルの指定と呼ぶ．ワイルドカードは正規表現とは異なるので注意すること．

ワイルドカードによる指定の方法は MS-DOS[18] とは異なり，シェルによるファイル名の置換という形で行なわれる．実際，次のような規則でファイル名の置換が行なわれる．

- *, ?, [, { のいずれかの文字を含んでいるか、あるいは ~ で始まるワードで，引用符で囲まれていないものはアルファベット順にソートしたファイル名のリストに展開される．
- * 先頭の '.' を除く（０個以上の）いずれかの文字に一致．
- ? 先頭の '.' を除くいずれかの１つの文字に一致．
- [...] 囲まれたリストまたは範囲内のいずれかの１つの文字に一致．リストは文字列．範囲は負の符号 (-) で区切った２つの文字で，すべての ASCII 文字を含む．
- {str, ...} コンマで区切ったリスト内で各文字列（またはファイル名の一致するパターン）に展開する．前述のパターンを一致させる式とは異なり，この展開結果はソートされない．たとえば，{b, a} は ‘b’ ‘a’ に展開されます （‘a’ ‘b’ ではない）．特殊な場合として，文字 { と } は文字列 {} とともにそのまま渡される．
- ~user 変数 home の値によって示されるホームディレクトリ，または user のパスワードエントリによって示されるそのパターン．
- *, ?, [...] だけがパターンマッチングを意味する．
- ファイル名がこれらを含むパターンに一致しなければ，エラーとなる．ピリオド （.）がファイル名かパス名の最初の文字である場合，明示的に一致しなければならない．スラッシュ（/）も明示的に一致しなければならない．

したがって，これらワイルドカードで利用される文字をファイル名に含むファイルを引数に指定する場合には，シェルによるワイルドカード展開を抑制するために " で囲むことが必要となる．

Example 4.4.3 あるディレクトリにあるファイルが

```
.a .ab a aa aaa ab bbb c d a.b
```

の１０個であったとする．この時，いろいろなワイルドカードでマッチするファイル名は以下の表のようになる．

[18] MS-DOS におけるワイルドカードは，コマンドインタープリタ (command.com) ではなく，システムコールで展開される．

ワイルドカード	.	..	.a	.ab	a	aa	aaa	ab	bbb	c	d	a.b
*	×	×	×	×	○	○	○	○	○	○	○	○
?	×	×	×	×	○	×	×	×	×	○	○	×
??	×	×	×	×	×	○	×	○	×	×	×	×
??*	×	×	×	×	×	○	○	○	○	×	×	×
a?	×	×	×	×	○	○	×	○	×	×	×	×
a*	×	×	×	×	○	○	○	○	×	×	×	×
a{a,b}	×	×	×	×	×	○	×	○	×	×	×	×
a{b,a}	×	×	×	×	×	○	×	○	×	×	×	×
[a-c]*	×	×	×	×	○	○	○	○	○	○	×	○
.	×	×	×	×	×	×	×	×	×	×	×	○
.?	×	○	○	×	×	×	×	×	×	×	×	×
.*	○	○	○	○	×	×	×	×	×	×	×	×

最後の２つの例では, ディレクトリ自身を示す ., 親ディレクトリを示す .. にも一致していることに注意. もし, .a, .ab のみに一致させたいときには, .[a-z]* 等を指定しなければならない. また, . が先頭にある場合と, そうでない場合にはワイルドカードのマッチの仕方が異ることに注意しよう.

Example 4.4.4 csh では, ~ がユーザのホームディレクトリを示すことを用いると, ホームディレクトリにある X というファイルは ~/X とすれば良いことがわかる. また, ユーザ名が user1 であるユーザのホームディレクトリは ~user1 とあらされる.

4.4.3 コマンドヒストリ

シェルを利用している場合に, 直前にシェルから実行したコマンドを再び実行する場面が多い. シェルは, 直前に実行したいくつかのコマンドを記憶しているので, それらを呼び出すことにより, 以前に実行したコマンドの入力が簡単になる. シェルが記憶している直前に実行したコマンドを**ヒストリ** (**history**) と呼び, history コマンドによって呼び出すことが可能である. を用いる. （なお, どのくらい以前のコマンドを記憶しているかは, シェルの設定に依存する.）

【利用法】 history

実際に history コマンドを実行すると,

```
% history↵
1  ls
2  ls -a
3  history
```

という出力を得る. 以下では csh でのヒストリの利用法をみてみよう.

この状態のもとで, ヒストリ番号を入力することにより, 過去に入力したコマンドを再現することが可能である. たとえば, 「ヒストリ番号 1」のコマンドを実行するには, !1 と入力する.


```
% !1↵
（ls が実行される）
% history↵
1  ls
2  ls -a
3  history
4  ls
5  history
```

また, !! は直前に実行したコマンドを表す.

```
% !!↵
（history が実行される）
% history↵
1  ls
2  ls -a
3  history
4  ls
5  history
6  history
```

以下同様に, 次のようなヒストリの利用が可能である. （より詳細なヒストリについては csh のオンラインマニュアルを参照.）

- !! : 直前のコマンド.
- !n : ヒストリ番号 n のコマンド.
- !-n : 現在のヒストリ番号の n 個前のコマンド. (!-1 が直前を表す)
- !str : str から始まる直前のコマンド. （上の例では !ls とすると, ヒストリ番号 4 が実行される.
- !?str? : str を含む直前のコマンド. （上の例では, !?s? とすると, ヒストリ番号 6 が実行される.

4.4.4 リダイレクトとパイプとジョブ制御

端末に接続されたシェルは[19]標準入力, 標準出力, 標準エラー出力の３つのデバイスを持っている（と考えよう）. 標準入力とは, 普通キーボードからの入力を指し, 標準出力と標準エラー出力は普通端末に結び付いている. しかし, それら標準入出力を切替えることによって, より便利な使い方ができる.

それら, それら標準入出力を切替える方法が, リダイレクトとパイプである. ここでは, これらのデバイスの切替を csh の場合に説明する.

4.4.4.1 リダイレクト

リダイレクトとは, 標準入出力や標準エラー出力をファイルに割り当てる方法である. 標準入力をファイルに割り当てるには < を, 標準出力をファイルに割り当てるには > を使う. 次の例はファイル file1 の内容を file2 に出力している.

```
cat < file1 > file2
```

この際, 既に file2 が存在している時には, file2 の内容はリダイレクトによって上書きされる. もし, file2 の末尾に追加したい時には, > の代わりに >> を用いる.

[19] 端末に接続されていない時でもいいのだが.

また, > または >> のかわりに >& または, >>& を使うと, 標準出力だけでなく, 標準エラー出力もリダイレクトすることができる.

Example 4.4.5 あるコマンド xx の出力をファイル file.out に記録するためには,

```
xx > file.out
```

とすればよい. この時, コマンド xx からエラー出力が行われると, それはターミナル（画面）に出力される.

Example 4.4.6 あるコマンド xx の出力とエラー出力をファイル file.out に記録するためには,

```
xx >& file.out
```

とすればよい.

Example 4.4.7 あるコマンド xx の出力をファイル file.out に, エラー出力をファイル file.err に記録するためには,

```
(xx > file.out) >& file.err
```

とすればよい. これは, ( ) で囲まれた部分を一つのコマンドとみなし, そのエラー出力をリダイレクトしていると理解すれば良い.

4.4.4.2 パイプ

パイプとは, 標準出力の内容を他のコマンドの標準入力に連結する方法で, これによって余分な操作を省くことができる. 次の例は, ファイルの内容を表示して, その結果を uniq というコマンドに渡している例である.

```
cat file1 | uniq
```

このように, 入出力のパイプには | を用いる.
また, | のかわりに |& を使うと, 標準出力だけでなく, 標準エラー出力もパイプすることができる.

Example 4.4.8 あるコマンド xx の出力を別のコマンド yy の標準入力にパイプするには,

```
xx | yy
```

とすればよい. この時, コマンド xx からエラー出力が行われると, それはターミナル（画面）に出力される.

Example 4.4.9 あるコマンド xx の標準出力とエラー出力を別のコマンド yy の標準入力にパイプするには,

```
xx |& yy
```

とすればよい.

Example 4.4.10 あるコマンド xx の標準出力を別のコマンド yy の標準入力にパイプし, xx と yy のエラー出力をファイル file.err に記録するためには,

```
(xx | yy) >& file.err
```

とすればよい.

Remark 4.4.11 パイプやリダイレクトの細かい制御は csh ではなかなか難しい. sh (Bourne shell) を用いることで, パイプやリダイレクトの制御を細かく行うことが可能になる. sh では, ファイルディスクリプタという概念があり, ファイルディスプリプタ番号 0 が標準入力に, 1 が標準出力に, 2 が標準エラー出力に割り当てられている.
sh で上の例の制御を行う場合には, それぞれ以下のようにすれば良い.

- 標準出力をファイル file.out に記録する.

  ```
  xx > file.out
  ```

- 標準出力と標準エラー出力をファイル file.out に記録する.

  ```
  xx 1> file.out 2>&1
  ```

  2>&1 により, ディスプリプタ 2 への出力を, ディスプリプタ 1 への出力に付随させることを示している. sh では「コマンドラインに記述された内容が右から左に」翻訳される. そのため, xx 2>&1 > file.out とすると, 先に標準出力が file.out に割り当てられ, その後に標準エラー出力が標準出力に付随させられる. したがって, 標準エラー出力の内容は file.out には書き出されない.

- 標準出力をファイル file.out に, 標準エラー出力をファイル file.err に記録する.

  ```
  xx 1> file.out 2> file.err
  ```

- 標準出力をパイプする.

  ```
  xx | yy
  ```

- 標準出力と標準エラー出力両方をパイプする.

  ```
  xx 2>&1 | yy
  ```

- 標準出力をパイプし, 標準エラーをリダイレクトする.

  ```
  xx 2> file.err | yy
  ```

- 標準エラーをパイプし, 標準出力をリダイレクトする.

  ```
  xx 2>&1 > file.out | yy
  ```

  この例を csh で実現することは困難である.

4.4.4.3 バックグラウンドでの実行とジョブ制御

UNIX ではシェルから入力したコマンドの終了を待っている状態を, 「プロセスが**フォアグラウンド** (**foreground**) で実行されている」といい, コマンドの終了を待つことなしに, 制御がシェルに戻るようにすることを, プロセスの**バックグラウンド** (**background**) 実行と呼ぶ. 即ち, 入力などを要求しないコマンドに対しては, その終了を待つこと無しに, 次のコマンドを入力できる. 次の例は, xclock をバックグラウンドで実行した例である.

```
xclock &
```

このように, バックグラウンドでの実行には & を指定すれば良い.
一旦バックグラウンドで実行したプロセスは, fg コマンドを用いることにより, フォアグラウンドに戻すことも可能である. ただし, 入出力を伴うプロセスをバックグランドで実行するときには, 標準入出力がスタックする可能性があるので, 標準入出力をリダイレクトしておくほうが安全である.

Example 4.4.12 あるコマンド xx と yy をバックグラウンド実行した後, xx をフォアグラウンドに戻す.
まず, 2つのコマンドをバックグラウンド実行する.

```
xx &
yy &
```

この後, jobs コマンドを実行してみよう. すると, jobs の出力は

```
[1] + Running    xx
[2] - Running    yy
```

となる．ここで，fg コマンドを実行すると，+ がついているジョブがフォアグラウンドに戻ってくる．ここでは，xx がフォアグラウンドに戻ってくるため，fg %2 として，ジョブＩＤを指定して fg コマンドを実行すれば，yy をフォアグラウンドに戻すことができる．

フォアグラウンドで実行しているジョブをバックグラウンドジョブにするためには，bg コマンドを利用することができる．

Example 4.4.13 コマンド xx を実行した後，これをバックグラウンドジョブにするには，一旦 xx の実行を中断する必要がある．そのため Control+ Z を入力すると，xx の実行が一時中断することができる．

この時点で jobs コマンドを見ると，

```
[1] + Suspended    xx
```

となっているので，bg とすれば，このジョブをバックグラウンドジョブとすることができる．

4.4.4.4 プロセスとジョブ制御

UNIX 上で動作している各アプリケーションなどは**プロセス** (**process**) と呼ばれ，UNIX カーネルで**プロセス・テーブル** (**process table**) と呼ばれる表によって管理されている．

4.4.4.4.1 キーボード入力によるジョブ制御 フォアグラウンドで実行されているプロセス，すなわち，シェルからコマンドラインで起動したプロセスは，**強制終了** (**force exit**) させたり，**一時停止**（**サスペンド suspend**）させることが可能である．そのためには，キーボードから以下の入力を行えばよい．

- 強制終了させたい場合： Control+C
- サスペンドさせたい場合： Control+Z

これらのキー入力は，シェルによってプロセスにシグナルを送るという形で実行させるが，全てのプロセスに対して望み通りの結果が得られるかどうかはわからない．

4.4.4.4.2 プロセステーブル フォアグラウンドで動いていないプロセスや，何らかの理由によって残ってしまった不必要なプロセスを強制終了させたりするにはどのような方法があるかを考えてみよう．そのためには，**プロセステーブル** (**process table**) と呼ばれるものを調べる必要がある．

プロセステーブルとは，そのＯＳ上で動作している全てのプロセスの一覧を表したデータであり，そこには各プロセスの所有者・プロセス名等が含まれている．プロセステーブルを表示するためには ps を用いる．

【利用法】 ps [-aAcdefjlLPy]

ps によって，現在のプロセス・テーブルを表示させることができる．オプションを何も与えなければ，ps を発行した端末に結び付いたプロセスのみを表示する．

すべてのプロセスを表示させるには，ps␣-ef とするのが最も単純である．この場合，

```
UID   PID  PPID  C    STIME TTY      TIME CMD
root    0     0  0   Mar 31 ?           0:02 sched
root    1     0  0   Mar 31 ?           1:21 /etc/init -
root    2     0  0   Mar 31 ?           0:00 pageout
root    3     0  1   Mar 31 ?         135:29 fsflush
naito 10346 28588  0 09:25:10 ?        0:00 twm
naito 10370 10346  0 09:25:11 ?        0:01 xscreensaver
```


という表が出力される[20]．ここで，

- UID とは，そのプロセスを実行しているユーザ名が表示され，プロセスの**実ユーザ**と呼ばれる[21]．
- PID とは，その**プロセスＩＤ**と呼ばれる，全てのプロセス区別する番号である．
- PPID とは，その**親プロセスＩＤ**である．**親プロセス**とは，各プロセスが制御を受けている直接の上位のプロセスのことである．

Remark 4.4.14 SYSV 系の UNIX では，システムカーネルはプロセステーブルにはあらわれてこない．そのかわり，プロセスＩＤが１番の init プロセスがシステム全体を統括しているように見せかけている．

また，一つのカーネルの下で動作できるプロセス数には限りがあり，その上限を越えて新しいプロセスを作成することは出来ない．近年の UNIX システムでは，一般ユーザのプロセス数にも上限が設定されていることが多い．

4.4.4.4.3 シグナル 各プロセスを強制終了させたり，一時停止させたりする場合には，各プロセスに対して**シグナル** (**singal**) を送るという操作が行われる．

シグナルとは，各プロセスに対するソフトウェア的な割り込み処理命令のことであり，特定のプロセスに対してシグナルを送るためのコマンドとして kill が用意されている．

【利用法】 kill [-signal] pid ...

プロセスＩＤ pid （複数指定可）にシグナルを送信する．

シグナルには多くの種類があり，どのシグナルを送るかによってプロセスの動きには違いがあらわれる．また，いくつかのシグナルを除き，シグナルを受取ったときの処理はプログラム内で変更可能であるため，必ずしも意図した通りの動きをするわけではない．

シグナルの種類	プロセスの一般的な動作	対応するキー入力
HUP	再起動または終了処理を行ってから終了	
TERM	終了処理を行ってから終了	
INT	終了処理を行ってから終了	Control+C
KILL	強制終了	
TSTP	一時停止	Control+Z

フォアグラウンドのプロセスに対してキーボード割り込みをかけることは，上のような特定のシグナルを送ることに等しい．

したがって，プロセスを確実に終了させるためには，

```
kill -KILL pid
```

というコマンドを発行すれば良いが，正常な終了処理を行わなかったり，**子プロセス** (**child prosess**) に対してシグナルを送信しなかったりするため，通常は，最初に INT シグナルを送り，それでも終了しない場合には KILL シグナルを送るという操作を行うべきである．なお，kill コマンドで -signal を省略した場合には INT シグナルが送信される．

なお，ログアウトするときには，意図的に残したプロセス以外は，すべて終了していなければならないが，プロセスがメモリーリークした場合などは，ウインドウが閉じてもプロセスが残っている場合があり，その

[20] ps は BSD 系ＯＳと SYSV 系ＯＳとでは，全く異なったオプション体系を持つ．
[21] UNIX では，プロセスの実ユーザ以外に**実効ユーザ**と呼ばれる概念もある．

場合には, プロセス・テーブルで残っているプロセスを探して, シグナルを送ることにより, プロセスを停止させなければならない. ログアウトしてもプロセスが残っていたりすると, 他のユーザがそのホストを利用できなくなったり, 意味もなく負荷の高いプロセスが残ったりすることがあるので, プロセスを残さないようにすることが重要である.

4.4.5 シェルの設定

4.4.5.1 コマンド・サーチ・パスと環境変数

csh などのシェルを利用して, コマンドを入力する際に, 入力されたコマンドが絶対パス, 相対パスで記述されていないとき, そのコマンドを検索するために, **コマンド・サーチ・パス** (**command search path**) という概念がある.

csh などのシェルには**環境変数** (**environment variable**) と呼ばれる変数が定義されており, その中に (csh の場合には) PATH という名前のついた環境変数がある. シェルの組み込みコマンド printenv を実行すると, 各種の環境変数が定義されていることがわかるが, PATH 環境変数は

```
/usr/sbin:/usr/ucb:/usr/bin:/usr/local/bin:/bin:/etc
```

などのように : で区切られたディレクトリ名が並んでいる. シェルから入力されたコマンドは, この順に検索され, 最初に見つかったコマンドが実行される[22].

セキュリティ上の問題から PATH 環境変数にカレント・ディレクトリを入れるのは望ましくない[23]. また, 各種環境変数は, シェルの起動時にシェルの初期設定ファイル (csh の場合には, ~/.cshrc) から読まれることが多い. したがって, 通常利用する環境変数は設定ファイルに記述するのが普通である.

シェルに設定されている環境変数の一覧を得るには, printenv を用いる.

【利用法】 printenv [ENV]

引数が省略されたときには, すべての環境変数を表示する. 引数があるときには, 引数に示された環境変数を表示する.

また, シェルに環境変数を設定するには setenv を用いる.

【利用法】 setenv VARIABLE VALUE

環境変数 VARIABLE に値 VALUE を設定する.

また, コマンドライン中では, 環境変数 VARIABLE は ${VARIABLE} として参照可能である. たとえば, PATH 変数の先頭に /a を追加するには次のようにすれば良い.

```
% setenv␣PATH␣/a:${PATH}↵
```

また, 設定された環境変数を消去するには unsetenv を用いる.

【利用法】 setenv VARIABLE

環境変数 VARIABLE を消去する.

なお, 環境変数は, シェルの子のシェルには自動的に引き継がれる.

[22] コマンドサーチパスを変更した場合や, コマンドサーチパス内に新しいコマンドが設定された場合には, rehash を用いて, コマンドサーチパスのハッシュテーブルを更新する必要がある.
[23] いわゆる「トロイの木馬」型のプログラムの実行を避けるためである.

4.4.5.2 初期設定ファイル

シェルにはログイン時に実行される**初期設定ファイル**が存在する. csh の場合には, ホームディレクトリにある .cshrc であり, ここでは, 環境変数やシェル変数の設定が行われる.

一般に, このような初期設定ファイルはホームディレクトリにあり, . で始まることが多い. これは, シェルのワイルドカード指定で * と指定しても, . で始まるファイルは対象外になり, ls コマンドで通常は表示されないという利点がある. シェルの初期設定ファイル中には, コマンドサーチパスをはじめとする, 各種の環境変数が定義されているだけではなく, コマンドエリアス等も定義されている.

シェルの初期設定ファイルは, **シェルスクリプト** (**shell script**) 形式と呼ばれる, 各シェルが解釈可能なテキスト形式のプログラムとして記述されている. 通常, 初期設定ファイルの内容はシェルの起動時に解釈され, 内容が反映されるが, 初期設定ファイルを変更した後, 既に起動しているシェルにその内容を反映させるには source を用いる.

【利用法】 source filename

filename で示されたシェルの設定ファイルの内容をシェルに反映させる.

初期設定ファイルはシェルスクリプトであるため, そのスクリプトを実行することで内容が反映されるように思えるが, スクリプトを実行すると, その内容はスクリプトを実行したサブ・シェルにのみ反映され, 起動側の親シェルには反映されない.

各自の csh の起動スクリプトの内容を書換えた時, その内容をシェルに反映させるには,

```
source ~/.cshrc
```

とすれば良いことがわかる.[24]

4.4.5.3 コマンドエリアス

後に述べるように, cp などの多くのコマンドには, 各種のオプションが用意されている. 例えば, cp コマンドで「上書きを行おうとする場合に, 確認メッセージを出す」というオプションとして, -i オプションがある. これらのオプションは, 毎回指定することが面倒であるが, これらのオプションを指定することで, ファイル操作などの安全性を高めることができる[25]. そのため, シェルには**コマンドエリアス** (**command alias**) と呼ばれる機能が用意されている. **エリアス** (**alias**) とは「別名」という意味[26]で, 複雑な名称に単純な名称を与える手法であると思って良い.

csh でコマンドエリアスを作成する最も単純な方法は, 初期設定ファイルに記述することである. 例えば, cp -i というオプション付きのコマンドを, 単に cp で起動したいときには,

```
alias cp 'cp -i'
```

とする. これによって, シェルで起動された “cp” という文字列 (正しくはトークン) は, 常に “cp -i” で置き換えられることになる. なお, alias を無効にするには,

```
unalias cp
```

[24] シェルの初期設定ファイルは, そのシェルによって解釈可能な形式になっている. したがって, 初期設定ファイルの内容を反映させるには, 一見すると csh filename としても良いように思える. しかし, これでは設定ファイルの内容は反映されない. 設定ファイルの内容が反映されるのは, シェルから呼び出された子シェルに対してであり, その子シェルは, 設定ファイルを実行して終了してしまうので, もとのシェルには設定ファイルの内容は反映されない.
[25] 不用意に既存のファイルを上書きしてしまわないため.
[26] メールの「アドレス帳」はエリアスのリストそのものである.

とすればよい. また, 一時的に（1回のコマンド入力時のみ）エリアスを無効にするには “\cp” として, “\” をつければ良い.

このエリアスがしていされている場合であっても, /usr/bin/cp と直接にパス名[27]を指定すれば, エリアスは有効にはならない. なぜなら, /usr/bin/cp は cp というトークンには一致しないからである.

4.5 UNIX の基本的なコマンド

以下では, UNIX の最も基本と思われるコマンドのいくつかを簡単に解説する. 詳しい解説はオンラインマニュアルを参照せよ. わからないことがあった場合には, まずマニュアルを見るという習慣をつけることが重要である.

以下では, <...> となっている部分は, ファイル, ディレクトリなどを指定することを示し, [...] は省略可能な指定を表す. ファイルなどを指定する時には, 複数のファイルを指定することができるが, 注意が必要である. また, a|b という表現は, a または b のどちらかか両方を指定できることを表す.

なお unix のコマンドは BSD 系または SYSV 系のいずれかによって微妙にオプションの取り扱いが異なることが多い.

4.5.1 ディレクトリの操作

ディレクトリを操作するコマンドとしては, cd, pwd, mkdir, rmdir などがある.

4.5.1.1 カレントディレクトリの変更

【利用法】 cd [<directory>]

カレントディレクトリを <directory> に変更する. <directory> の指定を省略した場合[28], ホームディレクトリを指定したことになる.

Example 4.5.1 カレントディレクトリを親ディレクトリに変更する.

```
% cd␣..↵
```

4.5.1.2 カレントディレクトリの表示

【利用法】 pwd

カレントディレクトリを表示するためのコマンドである.

4.5.1.3 新規ディレクトリの作成

【利用法】 mkdir [-p] <directory>

【オプション】

[27]コマンドサーチパス内にあるコマンドの絶対パスを知るためには which を使えば良い.
[28]MS-DOS では, cd とだけ入力すると, カレントディレクトリを表示することとなる. UNIX ではそのためには pwd を使う.

-p 存在しない親ディレクトリを作成してから <directory> を作成する.

指定する <directory> は既に存在するものであってはならない. また, -p オプションを指定しない限り, <directory> に指定したものの親ディレクトリが存在しない場合にはエラーとなる.

Example 4.5.2 オプション -p を指定しないと, ディレクトリ a が存在しない場合に a/b を作成しようとするとエラーとなる.

```
% mkdir␣a/b↵
mkdir:  Failed to make directory ‘‘a/b’’; No such file or directory
```

このような場合には, -p オプションをつければよい

```
% mkdir␣-p␣a/b↵
```

4.5.1.4 ディレクトリの削除

【利用法】 rmdir <directory>

削除するディレクトリは空でなくてはならない. 空でないディレクトリを削除するには rm を使う.

Example 4.5.3 ディレクトリ a の中が空でない場合には rmdir は使えない.

```
% mkdir␣-p␣a/b↵
% rmdir␣a↵
rmdir:  directory ‘‘a’’:  Directory not empty
```

このような場合には, rm -r コマンドを用いる

```
% mkdir␣-p␣a/b↵
% rm␣-r␣a↵
```

4.5.2 ファイルの操作

ファイルの操作, 表示などを行うコマンドには, ls, cp, rm, mv, cat, more, grep, diff などがある.

4.5.2.1 ディレクトリの内容の表示

【利用法】 ls [-aAcCdfFgilLqrRstu1] [<filename>]

<filename> を省略した場合には, カレントディレクトリのファイルの一覧を見ることができる. -aAcCdfFgilLqrRstu1 の部分はオプションと呼ばれ, 何を指定するかによって, ls の出力結果が異なる. 例えば, -l を指定した場合は, ファイルの属性, サイズなどの詳しい情報を得ることができる. しかし, 全てのオプションを同時に指定できるわけではなく, 排他的なものもあるので注意すること.

. からはじまる名前のファイルを表示させるには a オプションが必要である. <filename> として, ディレクトリを指定すると, そのディレクトリの中のファイルの一覧が表示される.

【オプション】

-a 全てのエントリを出力

-l 各ファイルについてモード, 所有者名等を出力

-g -l に加えてグループ名も出力

-F ファイルの内容を示す記号を末尾に追加

-t ファイルの最終修正時刻順に出力

Example 4.5.4 ディレクトリ内のファイルの一覧を得る.

```
% ls␣-algF↵
total 11
drwx------   3 naito    math          512 Mar  4 12:36 ./
drwxr-xr-x 163 naito    math         9216 Mar  4 12:36 ../
drwx------   2 naito    math          512 Mar  4 12:36 b/
```

- a オプションにより, . ではじまるファイルも表示させている.
- lg オプションにより, ファイルの所有者・グループも表示させている.
- F オプションにより, ディレクトリや実行形式などの種別も表示させている.

ls は BSD 系のＯＳと SYSV 系のＯＳでは, オプションの取り扱いが微妙に異なる. 特に l, g オプションの扱いが異るので注意すること.

Example 4.5.5 ディレクトリ内のファイルの一覧を得る. （オプションなし）

```
% ls↵
b
```

ここで ls␣b とすると, ディレクトリ b 内のファイル一覧を得ることができる.

4.5.2.2 ファイルをコピーする

【利用法】

【1】 cp [-i] <filename1> <filename2>

【2】 cp -r|R [-i] <directory1> <directory2>

【3】 cp [-i(r|R)] <filename> <directory>

1 の使用法の場合, cp は <filename1> の内容を <filename2> にコピーする. 2 の使用法の場合は, <directory1> のファイルを再帰的に <directory2> にコピーする. 3 の使用法の場合は, <filename> (複数指定可能) を <directory> にコピーする.

【オプション】

-i 対話モード

-R—**r** ディレクトリを再帰的にコピー

Example 4.5.6 ディレクトリ a 内に b, c というファイルがある場合, これをディレクトリ d にコピーする.


```
% cp␣a/b␣a/c␣d↵
```

Example 4.5.7 ディレクトリ a 内の全てのファイルを, 新規ディレクトリ d にコピーする.

```
% cp␣-R␣a␣d↵
```

4.5.2.3 ファイルの移動またはファイル名の変更

【利用法】 mv [-i] <filename1> <filename2>

mv はファイル名の変更や, ファイルの移動を行なう. ファイルとしてディレクトリも指定可能だが, ファイルシステムを跨がるディレクトリの移動はできない.

【オプション】

-i 対話モード

Example 4.5.8 ディレクトリ a 内に b, c というファイルがある場合, これをディレクトリ d に移動する.

```
% mv␣a/b␣a/c␣d↵
```

Example 4.5.9 ディレクトリ a 内の全てのファイルを, 新規ディレクトリ d に移動する. （ディレクトリ a をディレクトリ d という名前に書換えることと同値.）

```
% mv␣a␣d↵
```

Example 4.5.10 ディレクトリ a 内の全てのファイルを, a ディレクトリを残したまま, 新規ディレクトリ d に移動する.

```
% mv␣a/*␣d↵
```

Remark 4.5.11 Example 4.5.7, 4.5.9, 4.5.10 の違いに注意. Example 4.5.7 では -R オプションが必要であった.

4.5.2.4 ファイルまたはディレクトリの削除

【利用法】 rm [-fir] <filename>

<filename> で指定した 1 つまたは複数のファイルを削除する. ディレクトリを削除する場合は r オプションが必要.

【オプション】

-i 対話モード

-r ディレクトリを再帰的に削除

-f 書き込み不可であっても削除

Example 4.5.12 ディレクトリに a, aa というファイルがある場合, これらを削除する.

```
% rm␣a␣aa↵
```

または

```
% rm␣a*↵
```

後者はワイルドカードを用いている.

Example 4.5.13 ディレクトリにある *.c にマッチするファイルと, サブディレクトリ b にある *.c にマッチするファイルを削除する.

```
% rm␣*.c␣b/*.c↵
```

4.5.2.5 ファイルの出力

【利用法】 cat [<filename>]

cat は指定したファイルを内容を標準出力に書き出す. <filename> が省略された場合は, 標準入力から読みとる. <filename> には複数を指定することができる.

Example 4.5.14 ファイル foo を標準出力（画面）に出力する.

```
% cat␣foo↵
```

Example 4.5.15 ファイル foo を新規ファイル bar にコピーする.

```
% cat␣foo␣>␣bar↵
```

Example 4.5.16 ファイル foo0, foo1 の内容を, この順序で連結して, 新規ファイル bar に書き出す.

```
% cat␣foo0␣foo1␣>␣bar↵
```

Example 4.5.17 標準入力（キーボード）から入力を読み取って, その内容を新規ファイル bar に書き出す.

```
% cat␣>␣bar↵
aaa␣bbb
aaa␣bbb
aaa␣bbb
Control+d
```

ここで, Control+d は EOF という特殊な文字を表し, キーボードからの入力の終了を示す.

4.5.2.6 テキストファイルの表示

【利用法】 more <filename>

more はテキストファイルの内容を端末に 1 画面ずつ表示する. more コマンドを終了するには Q を入力すればよい.

more よりも高機能なテキストファイルを表示するコマンドである, less が搭載されているシステムもあるが, less は標準 UNIX コマンドではない.

4.5.2.7 グレップ

グレップ (**grep**) とは, テキストファイルの中に特定の文字列のパターンが存在するかどうかを調べることである.

【利用法】 grep [-i] [-v] <pattern> <filename>...

grep コマンドは, pattern で示された文字列パターンが, filename の中に存在するかどうかを調べる. もし, 存在すれば, そのパターンを含む行を出力する.

【オプション】

-i 大文字と小文字の区別を無視

-r <pattern> に含まれない行を出力

Example 4.5.18 ファイル a の中に aaa という文字列パターンがあるかどうかを調べる. ファイル a は,

```
aaa
aa
bb aaa
```

という内容であると仮定すると,

```
% grep␣aaa␣a↵
aaa
bb aaa
```

となる.

4.5.2.8 ファイルの差分

ファイルに対して**差分** (**difference**) を調べるとは, 2 つのファイルの違いを調べることである.

【利用法】 diff <file1> <file2>

Example 4.5.19 ファイル a と b の違いを調べる. それぞれのファイルは, 先頭の行を除いて一致しているとすると,

```
% diff␣a␣b↵
1c1
< a
---
> d
```

という出力を得る. これは 1 行目（から 1 行目まで）に違いがあり, 違いは, ファイル a には, 文字 a が, ファイル b には, 文字 d が違っていることを示している.

4.5.2.9 ファイルの総量

あるディレクトリ以下のファイルの総量を調べるには du コマンドを使う.

【利用法】 `du [-s] [-a] [-k] <file ...>`

【オプション】

-a 各ファイルに対してエントリを生成.

-s 指定したファイルそれぞれについての合計のみを表示

-k ブロック単位 (512 バイト) ではなく 1024 バイト単位で出力

Example 4.5.20 ユーザのホームディレクトリ以下のファイルの総量を調べる.

```
% su␣-s␣-k␣~↵
9 941 692 /home/staff/naito
```

この結果により, ホームディレクトリ以下には約 9.9 M バイトのファイルがあることがわかる.

4.5.3 ファイルのモードに関するコマンド

4.5.3.1 umask の表示と変更

【利用法】 `umask [<newumask>]`

Example 4.5.21 現在の umask 値の表示を行い, umask 値を 007 に変更する.

```
% umask↵
77
% umask␣007↵
% umask↵
7
```

4.5.3.2 パーミッションの変更

【利用法】 `chmod [-fR] <mode> file...`

【オプション】

-f シンボリックリンクのパーミッションを変更（デフォールトでは, リンク先のパーミッションが変更される.）

-R 再帰的にパーミッションを変更

mode は 8 進数値または記号で表し, g+r という表記は group に対して read 属性をつけることを意味する. go+X は user の実行属性があるファイルに対してのみ, group, other に対する実行属性をつけることを意味する.

Example 4.5.22 現在のサブディレクトリ以下の全てのファイルに対して, 誰もが読み出せるようにする. さらに, ディレクトリ及び実行ファイルの場合には execute 属性をつける.

```
% chmod␣-R␣go+rX␣*↵
```

ただし, この場合にはカレントディレクトリの属性は変更されていない.

4.5.4 オンラインマニュアル

オンラインマニュアルを参照するには, man というコマンドを使う.[29] 例えば, cp のオンラインマニュアルを見るには, man␣cp とする.

オンラインマニュアルは, UNIX のコマンドばかりではなく, システムコール, C の関数などの内容も含まれる. したがって, 目的によっては, 同じ名前のコマンドまたはシステムコールなどが存在するので, それらを区別する必要が出てくる. そのために, オンラインマニュアルはいくつかのセクションにわかれている.

1. ユーザコマンド
2. システムコール
3. ライブラリ
4. 装置とネットワークインターフェース
5. ファイルの形式
6. ゲームとデモ
7. 環境, 表と troff のコマンド
8. 保守用コマンド

このようなセクションにわかれているコマンドまたは関数などのオンラインマニュアルを見るためには,

【利用法】 `man [-s <section>] <command>`

という使い方をする.

Example 4.5.23 man␣printf とすると, printf コマンドのオンラインマニュアルが出てくるので, C の関数 printf のオンラインマニュアルを見るためには, man␣-s␣3s␣printf とする.

詳しいセクションの分類に関しては, 各セクションの intro のオンラインマニュアルを参照すれば良い.

なお, オンラインマニュアルをプリントすることは, 著作権法に触れる恐れがあるので, 注意すること.

演習問題

Exercise 4.5.24 カレントディレクトリにある file1.c, file2.c というファイルを, それぞれ file1.cc, file2.cc というファイルに名前を変更したい. この際,

```
mv␣*.c␣*.cc
```

としようとしたが, エラーがでた. 何故かを考えよ.

Exercise 4.5.25 カレントディレクトリにあるファイルの一覧を得ようとして,

```
ls␣*
```

としようとした. 望むような結果が得られるか？もし, 得られないのなら, 何故かを考えよ.

Exercise 4.5.26 UNIX のファイルシステムの仕組みを調べ, ファイルシステムを跨がるディレクトリの mv ができない（mv コマンドでサポートしない）理由を考えよ.

[29] man man としてみよ.

4.5.5 unix コマンドのまとめ

ここまでに示した unix のコマンドをまとめると以下の表の通りとなる.

ファイル操作	
ls	ディレクトリの内容表示
cp	ファイルのコピー
mv	ファイルの移動
rm	ファイルの削除
cat	ファイルの出力
more	テキストファイルの表示
less	テキストファイルの表示
grep	グレップ
diff	テキストファイルの差分
du	ファイル総量の表示
ディレクトリ操作	
cd	カレントディレクトリの変更
pwd	カレントディレクトリの表示
mkdir	新規ディレクトリの作成
rmdir	ディレクトリの削除
ファイルのモード関連	
umask	マスク値の表示と変更
chmod	ファイル所有権の変更
プロセス関連	
kill	シグナルの送信
ps	プロセステーブルの表示
その他	
history	コマンドヒストリの表示
man	オンラインマニュアルの表示

4.6 emacs の利用法

通常 UNIX の標準的なエディタは vi と呼ばれるものである. vi は初心者には使いにくいため, emacs と呼ばれる高機能エディタを利用することが多い. ここでは emacs の基本的な利用法をまとめておく.

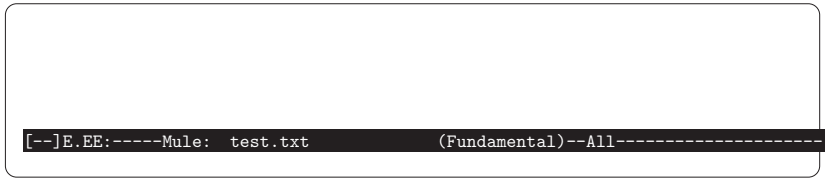
4.6.1 emacs

emacs の各種の機能は LISP を用いてコントロールすることができ, カーソル移動や, ファイルの保存機能なども LISP の関数として記述されている.

emacs の各種の操作は全て LISP の関数を呼びだすことで行われるが, それでは余りに大変なため, 初期設定ファイル `~/.emacs` 内で, 各種の LISP 関数とキー操作とを結び付け（**キーバインド** (**key bind**)）て操作を単純にしている. したがって, 良く行う操作については, 各自が初期設定ファイルにキーバインドを記述することで, 単純なキー操作でコマンドを実現できるようにすることも可能である.

4.6.1.1 emacs の起動方法

`emacs` とだけ入力すると, emacs が起動し, `*scratch*` という名前のついたウインドウ（emacs ではこれを**バッファ**(**buffer**) と呼ぶ）が開く. 特定のファイルを編集したい（または, 新規に作成したい）場合には, そのファイル名を引数に与えて emacs を起動すると, そのファイルの名前のついたバッファが自動的に作成され, そのファイルの編集モードに入ることができる.



この例は, `emacs text.txt` として実行したもので, バッファ名に `test.txt` となっていることに注意. 日本語環境で emacs を利用するために, mule と呼ばれる多言語対応の emacs を利用する.

このあと, 文字をキーボードから入力すると, **カーソル** (**cursor**) の位置に入力した文字が表示され, メモリ内に保存されていく. emacs バッファの下から２行めの反転した行を**モード行** (**mode line**) と呼び, ここにバッファの各種の情報が表示される. 最下行は**ミニバッファ**(**mini buffer**) と呼ばれ, emacs の各種機能を利用する際には, ここに入力が行われる.

4.6.1.2 カーソル移動

emacs でカーソルを移動する, すなわち編集するポイントを移動するには, 以下のコマンドを実行する.

左に１文字移動	C-b	一番先頭に移動	M-<
右に１文字移動	C-f	一番末尾に移動	M->
上に１行移動	C-p	１ページ下に移動	C-v
下に１行移動	C-n	１ページ上に移動	M-v

ここで `C-b` は Control+B とすることであり, `M-x` は ESC を押した後 X を押すことである. emacs のキー操作（**キーバインド** (**key bind**) ）を表すときには, 慣習的にこの表示を用いる. また, 初期設定ファイル `~/.emacs` に

```
(global-set-key "\eOA" 'previous-line)
(global-set-key "\eOB" 'next-line)
(global-set-key "\eOC" 'forward-char)
(global-set-key "\eOD" 'backward-char)
```

という記述を行うことにより, カーソルキーでカーソル移動を行うように設定することも可能である.

4.6.1.3 ファイルの保存・終了

emacs では編集中にファイルを保存して, 一旦 emacs を停止することも可能である.

emacs の終了	C-x C-c
バッファ内容の保存	C-x C-s
emacs の一時停止	C-z

emacs を一時停止すると, emacs はバックグランドで**停止状態** (**suspend**) になり, シェルに動作が戻る. シェルで fg コマンドを発行することにより, 再び emasc に戻ることができるので, いちいち emacs を終了しなくても, 編集中のプログラムコードをテストすることが可能である. ただし, ログアウト時には emacs は完全に終了しなければならない.

Example 4.6.1 emacs でCプログラムコードを書き, emacs をサスペンドしてプログラムコードをコンパイルする. その後, emacs を再び開く.

```
% emacs␣test.c↵                <==== test.c を emacs で開く.
----- emacs で編集して, Control+z でサスペンド
% gcc␣test.c↵                  <==== test.c をコンパイル.
----- コンパイル結果の表示
% fg↵                          <==== emacs に戻る.
```

このような操作をすれば, いちいち emacs を再起動しなくてもよい.
X Windows System 上の xemacs の場合には, xemacs をバックグラウンドで実行すればよい.

4.6.1.4 領域コピーとペースト

emacs ではいわゆる「カット&ペースト」の手続きがちょっと面倒である. emacs では, C-Space を押した場所（マークをつけるという）から, 現在のカーソル位置までが**領域** (**region**) とみなされる.

マーク	C-Space

このようにして指定した領域に対して, 以下の操作が可能である.

領域をカットしてメモリ内に保存	C-w
領域をカットせずメモリ内に保存	M-w
保存したカットバッファをペースト	C-y

保存した領域は「カットバッファ」に保存されるが, C-y を行うときにペースト（ヤンク）される内容は, カットバッファの先頭に入っている内容がヤンクされる. emacs では「カットバッファ」には複数の内容が保存されているため, 以前に「カットバッファ」に入れたものを取り出すことも可能である.

Example 4.6.2 emacs でカット&ペーストを行う最も単純な作業.
emacs で, バッファが以下の状態になっているとき, line 3 の行をカットして, line 4 の行の次の行にヤンクする.

```
This is a test for emacs line 1.
This is a test for emacs line 2.
This is a test for emacs line 3.
This is a test for emacs line 4.
```

1. 3行目 T の位置にマークを行う. （この位置で C-Space）
2. 4行目先頭に移動. （C-n C-a）

3. 4行目先頭で領域のカット. （C-w）
   この時, 次のようになっている.

   ```
   This is a test for emacs line 1.
   This is a test for emacs line 2.
   This is a test for emacs line 4.
   ```

4. 4行目の次の行に移動. （C-n C-a）
5. ヤンク. （C-y）

Example 4.6.3 emacs でコピー&ペーストを行う最も単純な作業.
emacs で, バッファが以下の状態になっているとき, line 3 の行をコピーして, line 4 の行の次の行にヤンクする.

```
This is a test for emacs line 1.
This is a test for emacs line 2.
This is a test for emacs line 3.
This is a test for emacs line 4.
```

1. 3行目 T の位置にマークを行う. （この位置で C-Space）
2. 4行目先頭に移動. （C-n C-a）
3. 4行目先頭で領域のコピー. （M-w）
   この時, 次のようになっている.

   ```
   This is a test for emacs line 1.
   This is a test for emacs line 2.
   This is a test for emacs line 3.
   This is a test for emacs line 4.
   ```

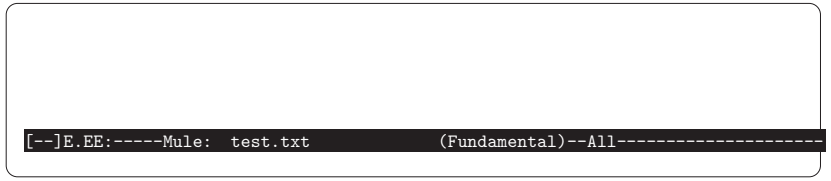
4. 4行目の次の行に移動. （C-n C-a）
5. ヤンク. （C-y）

4.6.1.5 複数バッファの制御とファイルのオープン

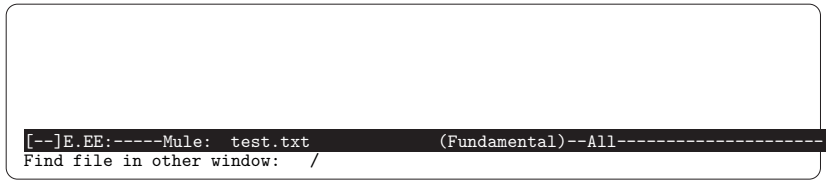
emacs では複数のバッファを同時に編集可能である.

ファイルを開く	C-x C-f （このあとにファイル名をミニバッファに入力）
上下にバッファを分割し, ファイルを開く	C-x 4 f （このあとにファイル名をミニバッファに入力）
上下にバッファを分割する	C-x 2
左右にバッファを分割する	C-x 3
現在のバッファを閉じる	C-x 0
他のバッファを閉じる	C-x 1
他のバッファに移動	C-o
他のバッファを開く	C-x b

Example 4.6.4 emacs でバッファを上下に分割して, 新しいファイルを読み込む.



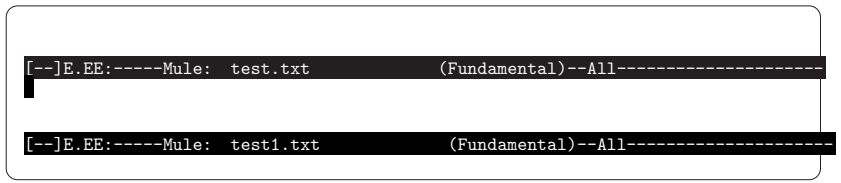
この状態で C-x 4 f と入力すると,



のようにミニバッファでファイル名の入力を求められる.

この時点で, 操作をやめたかったら C-g を押せば良い. いつでも, emacs の操作は C-g でキャンセルすることが可能である.

ここで, ミニバッファにファイル名を入力して ↵ を押せば,



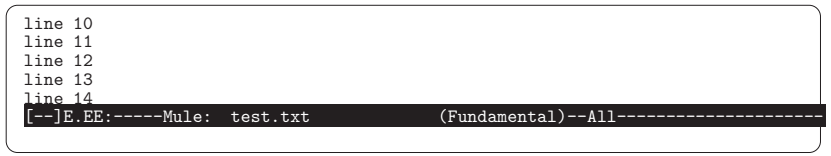
となり, バッファが上下に分割され, ファイルを読み込むことができる. この時「カレントバッファ」(カーソルの位置するバッファ) は新規ファイル側になる. カーソルを異るバッファに移動するには C-o を入力すればよい.

4.6.1.6 コマンドの入力

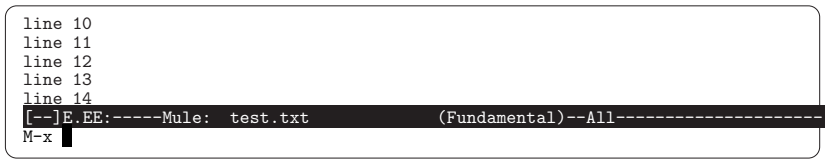
emacs ではキーバインドされていないコマンドを入力するときには, M-x を入力することで, いかなるコマンドも入力することが可能になる.

キーバインドのないコマンドで代表的なものは, goto-line 関数である. これは, バッファ内で指定の行に移動するコマンド (関数) である.

Example 4.6.5 emacs で指定の行に移動する.



この状態で M-x と入力すると,



のようにミニバッファで関数の入力を求められる. ここで, ミニバッファに goto-line↵と入力[30]すると,

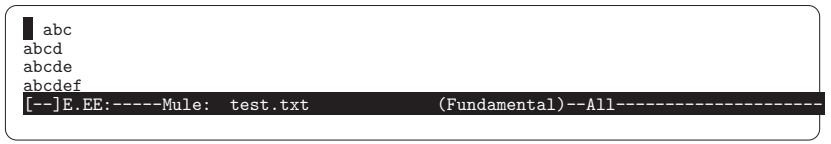
となり, 行番号の入力を求められるので, ジャンプしたい行番号を入力して ↵を押せばよい.

4.6.1.7 検索

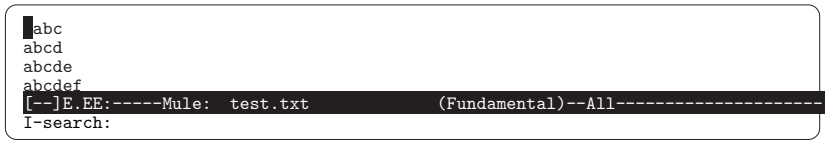
emacs でバッファ内の検索行うには, C-s または, C-r を用いる.

後方増分検索	C-s
前方増分検索	C-r

すなわち, 現在のカーソル位置からデータの後方に向かって検索を行う場合には C-s を, 前方に向かって検索を行うには C-r を用いる.



となっている場合に, C-s とすると,



となり, ミニバッファに I-seach と表示される. ここで検索したい文字列 (たとえば abcde) を入力すると, a を入力した時点で

[30] 実際には, goto-l まで入力した時点で Tab を入力すると, goto-line と表示される. これは, その時点で有効な関数名が goto-l まで一致するものが goto-line 以外に存在しないため, 自動的な**補完** (**completion**) が行われるためである.
また, goto- の時点で Tab を入力すると, *Compltions* バッファが開き, 候補の一覧を表示するという機能もある.

最初に見つかった a の後方にカーソルが移動する．この後, abcd まで入力すると，

```
abc
abcd
abcde
abcdef
[--]E.EE:-----Mule:  test.txt             (Fundamental)--All--------------------
I-search:  abcd
```

となり, 順に（インクリメントに）検索が行われることがわかる．検索を中止するには ↵ を入力すれば良い．

4.6.1.8 置換

emacs で置換を行うには, M-% を入力する．すると，

```
abc
abc
[--]E.EE:-----Mule:  test.txt             (Fundamental)--All--------------------
Query replace:
```

となり, ミニバッファに Query replace:　と表示され, 置換したい文字列（ここでは a としよう）を入力すると，

```
abc
abc
[--]E.EE:-----Mule:  test.txt             (Fundamental)--All--------------------
Query replace a with:
```

となり, 置換後の文字列の入力（ここでは b としよう）が求められる．すると, 置換の実行が始まり，

```
abc
abc
[--]E.EE:-----Mule:  test.txt             (Fundamental)--All--------------------
Query replace a with b:  (?  for help)
```

となり, 最初に一致した a で，「それを置換するかどうかを尋ね」られる．置換するならば ↵ を, 置換しないならば Space を押せば良い．

また, このように「置換するかどうか」を尋ねない文字列置換を行うには, M-x replace-string とすればよい．

4.6.2 emacs での日本語入力

emacs では（当然のことながら）日本語を入力することができる．emacs での日本語入力の方法にはいくつもの方法（種類）があり, その日本語入力メソッドの違いにより, 日本語入力の方法も異なっている．ここでは, emacs の日本語入力メソッドの代表的なものとして, Wnn または Canna を例にとり解説する．

4.6.3 emacs での日本語入力モードへの切り替え方法

emacs (Wnn または Canna) で日本語入力モードに入るには C-\ を入力する．（以下では Wnn が例となっているが, Canna でも大差はない.）すると，


```
[--]E.EE:-----Mule:  test.txt             (Fundamental)--All--------------------
```

となっていた「モード行」が

```
[あ]E.EE:-----Mule:  test.txt             (Fundamental)--All--------------------
```

と変化する.[31] このようにモード行の先頭が「あ」となっている状態が日本語入力モードを表している．逆に日本語入力モードから抜けるときにも C-\ を入力する．

4.6.4 emacs での日本語入力方法

日本語入力モードになっている場合には,「ローマ字入力」で日本語を入力することが可能である．この場合, “kyouhayoitenkidesu” とタイプすれば,

```
|きょうはよいてんきです|
[あ]E.EE:-----Mule:  test.txt             (Fundamental)--All--------------------
```

と表示され, | で囲まれた部分が現在のかな漢字変換の対象部分であることがわかる．かな漢字変換を実際に行うには C-SPACE を入力する．すると，

```
|きょうは 良い てんきです|
[あ]E.EE:-----Mule:  test.txt             (Fundamental)--All--------------------
```

となり,「きょうは」,「良い」,「てんきです」という 3 つの文節に分解され変換が行われたことがわかる．（空白によって文節が明示されていることに注意しよう.）この時, 次のようにして変換候補の変更, 文節の変更などができる．

次の候補	C-SPACE, C-n
前の候補	C-p
候補確定	↵
次の文節に変換対象を移動	C-n
前の文節に変換対象を移動	C-b
文節を短くする	C-o
文節を長くする	C-i

これらの操作を行った後,

```
|今日は 良い 天気です|
[あ]E.EE:-----Mule:  test.txt             (Fundamental)--All--------------------
```

となったら, ↵ を入力して結果を確定させれば良い．

なお, 人名などの単語を登録しておきたいときには, 登録対象の単語を「領域」として設定しておき, M-x toroku-region とすれば, 登録対象の単語の品詞などを指定した後に, ユーザごとの辞書に単語の登録を行うことができる．

[31] 場合によっては, 最下行に Input-method:　と表示されることがある．これは, 日本語入力メソッドが指定されていない状態で, その場合には適切な入力メソッドを入力する．何を指定してよいかわからない場合には Tab を入力すれば, 候補となる入力メソッドの一覧が表示されるはずである．

4.6.5 emacs コマンドのまとめ

ここまでに示した emacs のコマンドをまとめると以下の表の通りとなる.

左に１文字移動	C-b	
右に１文字移動	C-f	
上に１行移動	C-p	
下に１行移動	C-n	
一番先頭に移動	M-<	
一番末尾に移動	M->	
１ページ下に移動	C-v	
１ページ上に移動	M-v	
emacs の終了	C-x C-c	
バッファ内容の保存	C-x C-s	
emacs の一時停止	C-z	
emacs の終了	C-x C-c	
バッファ内容の保存	C-x C-s	
emacs の一時停止	C-z	
マーク	C-Space	
領域をカットしてメモリ内に保存	C-w	
領域をカットせずメモリ内に保存	M-w	
保存したカットバッファをペースト	C-y	
ファイルを開く	C-x C-f	ファイル名をミニバッファに入力
上下にバッファを分割し, ファイルを開く	C-x 4 f	ファイル名をミニバッファに入力
上下にバッファを分割する	C-x 2	
左右にバッファを分割する	C-x 3	
現在のバッファを閉じる	C-x 0	
他のバッファを閉じる	C-x 1	
他のバッファに移動	C-o	
他のバッファを開く	C-x b	
後方増分検索	C-s	
前方増分検索	C-r	
置換	M-%	(query あり)
	M-x replace-string	(query なし)
行番号を指定して移動	M-x goto-line	ミニバッファで行番号を指定
次の候補	C-SPACE, C-n	
前の候補	C-p	
候補確定	↵	
次の文節に変換対象を移動	C-n	
前の文節に変換対象を移動	C-b	
文節を短くする	C-o	
文節を長くする	C-i	
単語登録	M-x toroku-region	単語を領域として指定した後に実行する

References

[1] B. W. Kernighan and R. Pike. *UNIX* **プログラミング環境**. アスキー出版局, 1985.

[2] A. Frisch. *UNIX* **システム管理**. オライリージャパン, 1998.

[3] 坂本文. **楽しいＵＮＩＸ**. アスキー出版局, 1990.

[4] 坂本文. **続楽しいＵＮＩＸ**. アスキー出版局, 1993.