UNIXの基礎

渡辺 義明
watanaby@is.saga-u.ac.jp*

平成 27 年 4 月 6 日

1 はじめに

UNIX は、AT&T のベル研においてケン・トンプソンが 1969 年に作り始めたマルチユーザ・マルチタスクの OS である。1973 年頃にはトンプソンとデニス・リッチーによって C 言語に書き直された。C 言語記述であるため移植性が良いことと、ソースが大学等に安価に配布されたことから利用が広がった。特にカリフォルニア大バークレー校でビル・ジョイを中心に拡張された BSD 版が高機能であったため人気を呼んだ。また AT&T 自身が拡張した SYSTEM V 版が企業等に普及した。

このような UNIX の拡大に伴ってシステム間の互換性が下がった。その後、UNIX の満たすべき仕様として POSIX その他の標準が策定され、現状の UNIX はソースレベルでの互換性を保つようになってきている。

UNIX の商標権は現在 AT&T から The Open Group という団体に移っている。ここからライセンスを取得した OS だけが UNIX の商標を使える。例えば、Sun Solaris、IBM AIX、HP-UX 等がこれに当たる。これらに対して、現在 PC で良く利用されている Linux、FreeBSD 等は、UNIX の商標を使えない。BSD 系の FreeBSD、NetBSD、OpenBSD は、BSD 版から AT&T 由来のコードを取り除いた 4.4BSD Lite を元にして開発された。また Linux は、1991 年にリーナス・トーバルスがゼロから開発を始めた OS である。共に UNIX ライセンスは持たず、オープンソースとして普及している。ここでは、UNIX 互換である OS を含めて UNIX とする。現在 UNIX は、大型汎用機、ワークステーション、PC など各種コンピュータの OS として、また家電や携帯電話等の組み込み OS として広く使われている。

リーナス・トーバルスが開発したのは OS カーネルである。OS はその中核であるカーネルの他に多数のサービスプログラムを必要とする。それらの組合せ方や提供組織、使用目的などによって各種の Linux Distribution が存在する。リチャード・ストールマンは、フリーな UNIX 互換環境を作ることを目的とする GNU プロジェクトを 1983 年に開始し、コンパイラ gcc やエディタ emacs を始めとする多数のソフトウェアを開発した。これらは Linux Distribution の主要な要素となっている。GNU プロジェクトでは、コピーや改変が可能だがソース公開を要求する GPL ライセンスを提唱し、現在のオープンソースの発展の元を築いた。

PC に UNIX をインストールすると、X Window System と呼ぶグラフィックス環境が合わせてインストールされ、GUI 操作が可能になることが多い。この GUI 環境での操作は、MS Windows と

*このテキストは、平成 27 年 3 月まで本専攻の教授であった渡辺義明先生が書かれたものを、平成 27 年度担当の花田が一部改編したものである。

類似のものである。ここでは GUI 操作については触れず、より広範な環境で使えるキャラクタベースでの操作について、UNIX 上でのプログラム開発に必要な最低限の説明を行う。

2 利用の開始と終了

UNIX システムの利用を開始する処理を login、終了処理を logout という。login 手続きは環境によって様々であるが、ここでは MS Windows 上で動く PuTTY を使った SSH による遠隔 login について記す。PuTTY はネットワークからダウンロードしてインストールする。

起動時に表示されるダイアログで、ホスト名を入れる欄に、接続したい UNIX マシンの名前を指定する。総合情報基盤センターで提供している、学内から利用可能な UNIX マシンの名前を以下に示す。これらは全学共同利用のマシンである。過剰な負荷を掛けるなどして他の利用者に迷惑をかけないよう心がけること。

マシン名： ogi.cc.saga-u.ac.jp

「開く」をクリックすると開くウインドウに、ユーザ ID とパスワードを入れると利用できるようになる。「`logout`」または「`exit`」(シェル (4 節) を終了) で利用を終了する。

```
login as: useruser
????: passpass
[XX]$  ........
[XX]$ logout
```

PuTTY には各種の設定がある。日本語対応版で日本語を出すには、PuTTY のダイアログにおいて、「ウインドウ＞変換＞文字コードの設定＞ UTF-8」と文字セットを指定すること。また、ホスト名やフォントや色等の設定を行った後に、「セッション＞保存セッション」において、名前を入力して「保存」を押すと、以降はその名前を選ぶことで利用開始できる。

なお、PuTTY と同様な SSH クライアントソフトに ttssh もある。また GUI を使わず、キャラクタベースで遠隔ログインを行うには ssh または telnet コマンドが使える。「`ssh foo.bar.jp`」「`telnet foo.bar.jp`」のようにする。ただし最近はセキュリティ確保の点から、生のパスワードを流す telnet コマンドは許容されないことが多い。

他のホストとの間でファイルをコピーするには、ftp コマンドまたは scp コマンドを利用する。ただし ftp コマンドは生のパスワードが流れるため許容されないことが多い。scp コマンドの MSWindows 用 GUI プログラムに WinSCP があるが、これを使うには UNIX 側のホームディレクトリに以下の記述が必要である。

File: .bash_profile

```
test -n "$SSH_CLIENT" -a -z "$SSH_TTY" && export LANG=C
```

File: .login

```
if($?SSH_CLIENT && ! $?SSH_TTY) setenv LANG C
```

3 ファイルシステム

3.1 ファイル

UNIX のファイルは大きく分けて、通常のファイル、ディレクトリファイル、スペシャルファイルの 3 種類がある。通常のファイルは文書や実行プログラムの保存などに使われる。ディレクトリファイルは、ツリー状のファイルシステムのノードに当たるもので、そのノード下に存在するファイルの一覧表を登録したファイルである。スペシャルファイルは、入出力デバイス（キーボード、画面、フロッピーディスク、プリンター等）、その他をファイルとして扱うものである。

3.2 ディレクトリ

UNIX のファイルシステムは木を逆さにしたような構造に構成されている（代表的な構造を 8 節に示す）。根に当たるところを「ルートディレクトリ」と言い、「`/`」で表す。Windows のように、ドライブごとの構成ではなく、複数ドライブも一つの木構造に集約される。

ユーザー毎に割り当てられ、ログインの時に位置付けられるディレクトリを「ホームディレクトリ」と言う。例えば `user1` さんのホームディレクトリは通常`/home/user1` のように設定されている（上記のセンターマシンは、ユーザが非常に多数のため多階層になっている）。現在作業中のディレクトリをワーキングディレクトリまたは「カレントディレクトリ」と言う。ログイン時点ではホームディレクトリがカレントディレクトリであるが、コマンドにより移動できる。

3.3 基本的ファイル操作コマンド

以下にファイル操作の基本的コマンドを示す。

基本コマンド	機能
ls	ディレクトリの内容を出力 (list)
ls -l	ディレクトリの内容を詳細情報と共に出力 (list -long)
cat file1	ファイル file1 の内容を出力/連結 (catenate)
more file1	ファイル file1 の内容をページ毎に出力
cd path1	パス path1 へカレントディレクトリを移動 (change directory)
cp file1 file2	ファイル file1 から file2 へのコピー (copy)
mv file1 file2	ファイル名を file1 から file2 へ変更/移動 (move)
rm file1	ファイル file1 の削除 (remove)
mkdir dir1	ディレクトリ dir1 を作成 (make directory)
rmdir dir1	ディレクトリ dir1 を削除 (remove directory)

3.4 ファイル操作例

上で示した基本コマンドの簡単な実例を示す。

```
cd ..          カレントディレクトリを一つ上位のディレクトリへ移動する
ls             そこのファイル一覧を見る
cd /           カレントディレクトリをルートディレクトリへ移動する
cd home        カレントディレクトリを一つ下の home ディレクトリへ移動する
                /を最初に付けないと、カレントディレクトリからの相対位置での
                指定になる
cd /home/user   /を最初につけると、ルートディレクトリからの絶対位置
                指定になる
cd             ホームディレクトリに戻る
mkdir work     カレントディレクトリの下にディレクトリ work を作る
cp t.c work    ファイル t.c を、ディレクトリ work へコピーする
cd work        ディレクトリ work へ移動する
ls ..          一つ上のディレクトリのファイル一覧を見る
rm t.c         ファイル t.c を削除する（ここで消したのは work 中のファイル）。
cd ..          一つ上に移動する
rmdir work     ディレクトリ work を削除する
```

一つ上位のディレクトリ「..」の他にも、ホームディレクトリ「~」、カレントディレクトリ「.」を使った指定もできる。ファイル名入力は [TAB] キーを押すことで補完できる（ab[TAB] と入れると ab で始まるファイルが一つだけならファイル名全体 abcde.c が補完される）。

3.5 ファイルとディレクトリのアクセス権

ファイルやディレクトリについては、3 つのユーザカテゴリ毎に、3 つの権限が設定される。

3 つのユーザ・カテゴリ

- u: ファイルを所有するユーザ
- g: ファイルを所有するグループに属するユーザ
- o: それ以外のユーザ

3 つの権限

- r: 読み出し (通常ファイル：表示、ディレクトリ：内部ファイルの一覧表示)
- w: 書き込み (通常ファイル：修正、デイレクトリ：内部ファイルを追加、削除)
- x: 実行 (通常ファイル：コマンドとして実行、ディレクトリ：内部ファイルをアクセス)

これらのアクセス権は、「ls -l」において、「-rwxr-x---」のように表示される。最初の 1 文字はファイルタイプで、「-」は通常ファイルを、「d」はディレクトリを意味する。その他のタイプもある。

```
XX[1] ls -l
-rw-r--r--   1 watanaby se          2475  1月 31日 2007年 ldap.c
-rw-r--r--   1 watanaby se          2841  2月  2日 2007年 ldaps.c
-rw-r--r--   1 watanaby se          2842  2月  2日 2007年 ldaps.c~
-rwxr-xr-x   1 watanaby se          5544 11月 21日 2006年 prog
-rw-r--r--   1 watanaby se           143 11月 21日 2006年 prog.c
drwxr-xr-x   2 watanaby se          4096  8月 20日  15:27 test
```

続いて、ユーザ (u)、グループユーザ (g)、その他ユーザ (o) の権限が 3 文字ずつ計 9 個並ぶ。上の例では、所有ユーザは読み (r)・書き (w)・実行 (x) が可能、グループユーザは読み (r)・実行 (x) が可能、その他は全て不可を意味する。さらに、setuid ビット、setgid ビット、sticky ビットという特別なアクセス権ビットがある。

アクセス権の変更をするには、以下のように「chmod」を使う。

```
chmod 644 *.c    全 C ファイルの権限を 644=110100100=rw-r--r--にする。
chmod ug+x foo   ファイル foo に所有者 (u) とグループ (g) の実行権 (x) を加える。

chmod u=rw,go=r foo ファイル foo のアクセス権を rw-r--r--にする。
```

「644」の数字表記は、権限「rw-r--r--」を 2 進数「110100100」と考えて、3 つずつまとめて 10 進表記「644」したものである。

また、所有者とグループの変更は以下のように行う。

```
chown user1 x.c   x.c ファイルの所有者を user1 にする。
chgrp grp1 x.c    x.c ファイルのグループを grp1 にする。
```

4 シェルの基本機能

シェルは、UNIX においてユーザとのインターフェースを担うプログラムである。ユーザからのコマンドを受付けて解釈し、その実行をカーネルに依頼する役割を持つ。OS の外郭をなす貝殻 (Shell) という意味である。UNIX においては多くのシェルが作られており、少しずつ機能が異なる。以下では bash を基本として説明する。

4.1 メタ文字

複数のファイル名を一度に指定したい時などに便利な記号として、下のようなメタ文字がある。

メタ文字	機能
*	任意の長さの任意文字列に対応
?	任意の 1 文字に対応
[文字列]	[ ] 内の任意の 1 文字に対応
[文字 1 -文字 2]	文字 1 から文字 2 の間の 1 文字に対応
{文字列,...}	{ } 内の各文字列に対応

これらは次のように使用する。

```
ls *.c          末尾が「.c」となるファイル名を表示（t.c,test.c,..)
ls 9?s401       ２番目が任意文字のファイル名表示（90s401,91s401,..)
ls 90s40[123]   最後が 1 か 2 か 3 のファイル名表示（90s401,90s402,90s403）
ls [a-k]*       最初が a から k の間の文字で始まるファイル名（abc,dx.c,..）
ls {tst,test}*  tst または test で始まるファイル名表示（tst1.c,testprt,..）
```

これらのメタ q 文字はシェルで解釈され、具体的ファイル名に展開されて、コマンドに渡される。これに対し、MSDOS、MSWindows では展開せず「*」 のままコマンドに渡して、コマンド自身が解釈する。その概念で使うと間違うことがあるので注意すること。例えば cp、mv などの書き出し先指定に「*」 を使うとファイルを破壊する。

4.2 リダイレクションとパイプ

UNIX のコマンドは、データ入力を「標準入力」から受けとり、出力を「標準出力」へ出すことを基本にする。普通にログインした状態では、「標準入力」はキーボードに、「標準出力」はディスプレイに割り当てられている。この入出力を他のファイルへ切替えることをリダイレクションと言い、コマンドの入力をファイルから行なったり、出力をファイルに行なったりできる。入力切替えは「<」を、出力切替えは「>」を指定ファイル名の前に付けて、次のように行なう。ここで foo は、標準入力から入力を受けて、標準出力に出力する任意のコマンドと仮定する。

```
foo                       入力：キーボード   出力：ディスプレイ
foo <test.dat             入力：test.dat     出力：ディスプレイ
foo >test.out             入力：キーボード   出力：test.out
foo <test.dat >test.out   入力：test.dat     出力：test.out
```

これを利用すれば、エディタなしでも小さいファイルを次のようにして作成できる。ここで、cat は指定したファイルそれぞれの内容を標準出力へ書き出すコマンドであり、ファイルが一つも与えられないと標準入力から読み込む。

```
cat >newfile
This is the content of first line
This is second line
And this is last line
^D       (Control キーを押したまま、D キーを押す＝入力終了)
```

また、自作プログラムの出力をファイルに保存する時などにも便利である。

あるコマンドの標準出力を、別のコマンドの標準入力に接続することができる。これをパイプと言い、「|」でコマンドをつなげて次のように指示する。

```
ls x* | more        x で始まるファイル名の一覧を 1 ページ毎に止めながら見る。
```

4.3 コマンドの実行例

UNIX には、上で示したコマンド以外にも、極めて多数のコマンドがある。それらを組み合わせて複雑な処理を実行することもできる。以下に簡単なコマンドの例を示し、代表的コマンドを 6 節に示す。

```
ls -a                          ディレクトリの内容を隠しファイルを含め表示する
ls -l                          ディレクトリを詳細表示する
ls -la                         ディレクトリ全内容を詳細表示する
chmod 644 foo                  ファイル foo を全員が読み可能、
                                 自分だけ書き可能とする。
chmod 755 foo                  上記に加えて全員が実行可能とする。
ps -Af | more                  プロセス一覧詳細をページ単位で表示
ps -Af | grep 'pattern'        pattern を含むプロセスを表示
grep 'pattern' *.c             C ファイルの中で pattern を持つ行を表示
tar -cvzf foo.tar.gz bar       bar 以下を、アーカイブ foo.tar.gz に圧縮する
tar -xvzf foo.tar.gz           アーカイブを解凍する
find ./ -name 'pattern' -print 名前に pattern を持つファイルを表示
```

4.4 ヒストリ

過去に実行したコマンドを呼び出して実行することができる。これをヒストリ機能と言う。主な機能を下に示す。

コマンド/置換指示	機能
`history`	過去に実行のコマンドの一覧表示
`!!`	直前に実行したコマンドの再実行
!番号	指定番号のコマンド再実行
!文字列	指定文字列で始まる最近のコマンド再実行
^文字列 1 ^文字列 2	直前のコマンドを文字列 1 から 2 へ置換実行
↑	直前のコマンドを表示

4.5 特殊なキーストローク

UNIX 環境では、以下のキー入力は特殊な意味を持つ。

キー入力	意味
`Ctrl+U`	行先頭からカーソル位置までを削除する。
`Ctrl+H`	カーソルの前の文字を削除する (=BS キー)。
`Ctrl+D`	入力を終了する。シェルであれば終了する。
`Ctrl+C`	起動中のプログラムを終了する。
`Ctrl+Z`	一時的にプログラムを停止する（バックグラウンドジョブに入る）
`Ctrl+S`	スクリーン出力を停止する。
`Ctrl+Q`	スクリーン出力を再開する。

4.6 パス

コマンド名を入れると、対応する実行ファイルを環境変数 PATH に指定されたディレクトリから探す。PATH に入っていないディレクトリ中のファイルを実行するときは、絶対または相対ディレクトリの指定が必要である。カレントディレクトリが PATH に入っていない場合に、カレントディレクトリ中の mycmd を動かすには、カレントディレクトリを意味するドットをつけて

```
./mycmd
```

とする。どのディレクトリのコマンドが実行されるか知るには「`which cmd`」、環境変数の確認は「`printenv` または `set`」でできる。環境変数の設定方法は、シェルによって様々である。bash の場合は、「`export NAME=VALUE`」で、`NAME` に `VALUE` が設定できる。また、それまでの値に追加するには、「`PATH=PATH;~/mybin`」のように、従来の PATH に~/mybin（例）を加え改めて PATH と設定する。環境変数と類似のものに、シェルでのみ通用するシェル変数がある。

5 正規表現

4.3 節の `pattern` のようなところには、正規表現が使えることが多い。正規表現は、パタンマッチを記述するためのものであり、通常のテキスト文字と特殊な意味を持つメタ文字からなる。BRE（basic regular expression、基本正規表現) と ERE(extended regular expression、拡張正規表現) の 2 つ

の形式があり、ツールによって、どちらを使うかが決まる。また、`find` のように、シェルのメタ文字（4.1）を使うツールもある。

- BRE を使用するツール：grep, ed, sed, vi, emacs
- ERE を使用するツール：egrep, awk, perl

ERE では以下のメタ文字を使用する。

`c`	非メタ文字「`c`」にマッチ
`\c`	文字「`c`」にマッチ（メタ文字も通常文字とみなしてマッチ）
`.`	全文字にマッチ
`^`	文字列の先頭にマッチ
`$`	文字列の末尾にマッチ
`\<`	単語の先頭にマッチ
`\>`	単語の末尾にマッチ
`[abc...]`	「`abc...`」のいずれかの文字にマッチ
`[^abc...]`	「`abc...`」以外の文字にマッチ
`r*`	`r` で認識されるパタンの `0` 回以上の繰り返しにマッチ
`r+`	`r` で認識されるパタンの `1` 回以上の繰り返しにマッチ
`r?`	`r` で認識されるパタンの `0` 回または `1` 回の出現にマッチ
`r{m,n}`	`r` で認識されるパタンの `m` 個以上、`n` 個以下の並びとマッチ
`r1|r2`	`r1` または `r2` のいずれかにマッチ
`(r1|r2)`	上記をひとまとめの正規表現とする

BRE では、

```
+ ? ( ) { } |
```

は通常のテキスト文字であり、代わりにバックスラッシュでエスケープされた以下の文字を、メタ文字として使用する。

```
\+ \? \( \) \{ \} \|
```

下は、perl(ERE 記述) における文字列置換例である。

```
$ echo zzz1abc2efg3hij4 | perl -pe 's/(1[a-z]*)[0-9]*(.*)$/$2===$1/'
zzzefg3hij4===1abc
```

echo で出力した文字列を、パイプを経由して perl に渡して置換している。2 行目は結果出力である。ここで perl のコマンドである `s/pat1/pat1/` は `pat1` を `pat2` に置換することを意味する。例における `pat1` のところ「`(1[a-z]*)[0-9]*(.*)$`」は、以下を意味する。「(1 の後に小文字がゼロ個以上あり)、その後に数字がゼロ個以上あり、(その後に何かがゼロ個以上あり)、その後が末尾になっているパタン」。`pat2` の部分「`$2===$1`」は以下を意味する。「置換前パタンにおける 2 番目の括弧の中身（`$2`）があって、その後に`'===’`があって、その後に 1 番目の括弧の中身（`$1`）が続くパタン」

同じことを行う sed(BRE 記述) における文字置換例を次に示す。この場合、メタ文字として使うために、いくつかのバックスラッシュエスケープが挿入されている。また sed では、n 番目の括弧の中身を表すのに$n ではなく\n を使用する。

```
$ echo zzz1abc2efg3hij4 | sed -e 's/\(1[a-z]*\)[0-9]*\(.*\)$/\2===\1/'
zzzefg3hij4===1abc
```

さらに、リダイレクトを組み合わせれば、ファイルの中身を一括変換することができる。なお、上記では、1 行につき一回ずつの置換となるが、「g」を最後に付けることで 1 行にパタンを複数回適用することも可能である。より詳細な使い方は調べること。

6 man コマンド

コマンド等の使い方が分からないときは、オンラインでマニュアルを見ることができる。例えばコマンド awk の使い方は、「man awk」 で表示される。代表的コマンドを次ページに示すので、使い方は man コマンドで見ること。man コマンドでの表示はスペースキーで先へ、b キーで後ろへ進み、q で終了する。マニュアルには、そのコマンドと関連の項目が書かれていることがある。例えば printf(3V) とあれば、「man 3 printf」 でその関連項目のマニュアルを表示できる。コマンドと関数などに同じ名前が付けられているときは、最初に見つかった方が表示される。このような時は上のようにセクション番号を付加して man を呼び出せば他の項目を表示できる。なお、セクション番号は 1=ユーザコマンド、2=システムコール、3=ライブラリ関数、などと分類して付けられる。

コマンド	主な役割
alias	別名の割り当て
awk	パターン操作言語
bg	バックグラウンド実行
cal	カレンダーの表示
cat	ファイル連結/出力
cd	ディレクトリの変更
chgrp	ファイルグループを変更
chmod	ファイルパーミッションを変更
chown	ファイルオーナーを変更
clear	画面クリア
cmp	ファイルの比較
cp	ファイルのコピー
csh	C シェルの起動
date	日付の設定と参照
df	ファイルシステムの情報
diff	ファイルの差を出力
du	ディスクの使用状況表示
echo	指定文字列を出力
emacs	エディタ
exit	シェルを終了
fg	フォアグラウンド実行
find	ファイルの探索
finger	ユーザー情報
ftp	ファイル転送
g++	GNU C++コンパイラ
gcc	GNU C コンパイラ
gdb	デバッガ
grep	パターン探索
gzip	ファイル圧縮/解凍
head	ファイルの先頭出力
history	コマンド履歴出力
hostname	ホスト名の表示と設定
indent	C プログラムの整形
jobs	ジョブ一覧表示
kill	プロセスを停止
ln	リンクを張る
logout	ログインシェル終了
lpq	プリント出力待ち表示
lpr	プリンター出力
lprm	プリント出力待ち削除

コマンド名	主な役割
ls	ディレクトリ内容表示
mail	電子メール受発信
make	コンパイル・インストール自動化
man	マニュアル表示
mkdir	ディレクトリ作成
mount	ファイル構造をマウント
more	テキストファイル表示
mv	ファイルの移動
netstat	ネットワーク機能の状態表示
od	ファイルのダンプ
patch	ファイルを更新
perl	スクリプト処理言語
ps	プロセスの状態表示
pwd	現ディレクトリ表示
rm	ファイルの削除
rmdir	ディレクトリの削除
scp	セキュアな遠隔コピー
sed	ストリームエディタ
set	シェル変数の設定
setenv	環境変数の設定
sh	UNIX 標準シェルを起動する
sort	ファイルのソート
ssh	セキュアな遠隔端末操作
split	ファイルの分割
stty	端末オプションを設定する
su	ユーザを切り替える
tail	ファイル末尾出力
tar	ファイルの保存/復旧
telnet	リモートシステムへログイン
time	実行時間の出力
touch	タイムスタンプ変更
tr	文字変換
uniq	重複行を消す
uuencode	バイナリをテキストに
uudecode	uuencode から復元
w	ログインユーザ表示
wc	ファイル行数等計数
which	コマンド所在パス表示
who	ログインユーザ表示
passwd	パスワードの変更

7 プロセス

UNIX 上では、多数のプロセスを独立して動かせる。一つのプロセスが異常を起こしても、即システム全体の停止にはならない。プロセスは、端末につながった通常の「フォアグラウンド実行」と、端末から切り離されて動く「バックグラウンド実行」、それに「一時停止状態」を切替えられる。バックグラウンド実行はコマンドの最後に`&` をつけて、`testjob &` のように指示する。フォアグラウンド実行中のプロセスは`^Z` で一時停止状態になる。一時停止プロセスは、`fg` でフォアグラウンドに、`bg` でバックグラウンドの実行に移る。

また `ps` コマンドで現在動いているプロセス（特に指示しなければ自分の分だけ。全て表示は「`ps -Af`」）で表示できる。その表示中のプロセス番号を用いて、`kill` でプロセスを殺すことができる。ただし、間違って重要なプロセスを殺すと厄介なので、番号は間違わないように注意すること。

```
$ ps          <= プロセス一覧の表示
  PID TT STAT  TIME COMMAND
 5023 p0 IW    0:00 -sh (csh)
 5025 p1 S     0:00 -sh (csh)
 5159 p1 R     0:05 testjob
 5163 p1 R     0:00 ps
$ kill 5159   <= コマンド testjob を殺す
```

8 UNIX のディレクトリ構造

Linux を始めとする UNIX ベース OS の標準的なディレクトリ構成を以下に示す。例えば、ホスト固有の各種設定は/etc に、ユーザの書き込むデータは/home にまとめられている。別ドライブは、どこかの枝の先に mount コマンドでつなぐことで単一ツリーに集約される。ネットワークの先にあるドライブをマウントもできる。Windows のように複数ツリーに分離しない。

また、Windows のショートカットや MacOS のエイリアスのようなリンクファイルを ln コマンドで作成できる。ハードリンクとシンボリックリンクという 2 種類がある。

- / : ルート（根）ディレクトリ
  - bin : 全ユーザが使用する重要なコマンド
  - boot : ブートローダーの為の静的なファイル
  - dev : デバイスファイル
  - etc : ホスト固有のシステム設定
  - home : ユーザのホームディレクトリ
  - lib : 重要な共有ライブラリとカーネルモジュール
  - media : リムーバブル・メディアのためのマウントポイント
  - mnt : ファイルシステムの一時的なマウントポイント
  - opt : 追加アプリケーションのソフトウェア・パッケージ
  - root : root ユーザのためのホームディレクトリ
  - sbin : 重要なシステムコマンド
  - srv : システムによって提供されるサービスデータ
  - tmp : 一時的なファイル
  - usr : 共通的アプリケーションのための領域
    - /X11R6 : X Window システム
    - bin : ユーザが使用する大部分のコマンド
    - include : 標準的なインクルードファイル
    - lib : プログラミングのためのライブラリとパッケージ
    - local : ホスト固有のアプリケーションのための領域
    - sbin : 重要ではない標準的なシステムコマンド
    - share : アーキテクチャに依存しないデータ
    - src : ソースコード
  - var : 可変的データのための領域
    - cache : アプリケーションのキャッシュデータ
    - lib : 可変的な状況情報
    - lock : ロックファイル
    - log : ログファイルとディレクトリ
    - mail : ユーザのメールボックス
    - opt : /opt の為の可変的なデータ
    - run : ランタイムの可変的なデータ
    - spool : アプリケーションのスプールデータ
    - tmp : システム再起動時の為に予約されている一時的なファイル

9 vi エディタ

vi エディタもしくはその拡張である vim エディタは、殆ど全ての UNIX 環境に用意されている標準エディタである。どのような UNIX 環境にも対応できるように、基本的機能は覚えておくことを薦める。

9.1 基本的な操作

- 立ちあげ

  `vi File` File の編集開始

- 編集モード・コマンドモードの切替

  `i(insert)` カーソルの前に挿入へ
  `a(append)` カーソルの後ろに挿入へ
  `I` 行先頭に挿入へ
  `A` 行末尾に挿入へ
  `o` カーソルの下に挿入へ
  `O` カーソルの上に挿入へ
  `[esc]` コマンドモードへ

- 終了

  `ZZ` ファイルをセーブして終了
  `:q!` ファイルをセーブせずに終了
  `:wq` ファイルをセーブして終了

  キー入力文字がそのまま入力されるようなら編集モードにあるので、[esc] キーでコマンドモードに切り替えてから、`:q!` などとする。

9.2 基本的な編集操作

- カーソル移動

  `l` 1 文字前進（エル）
  `h` 1 文字後退
  `k` 前の行へ
  `j` 次の行へ
  （以上 4 つのキーはキーボードホームポジションに横並び）
  `w` 1 ワード前進 (forward)
  `b` 1 ワード後退 (backward)
  `0` 行頭へ (zero)
  `G` ファイル末尾へ
  `1G` ファイル先頭へ
  `nG` ファイル n 行目へ

- 画面スクロール

  `Ctrl+f` 1 スクリーン先へ
  `Ctrl+d` 半スクリーン先へ
  `Ctrl+e` 1 行先へ
  `Ctrl+b` 1 行後ろへ
  `Ctrl+u` 半スクリーン後ろへ
  `Ctrl+y` 1 スクリーン後ろへ

- コマンドの繰り返し

  `nCmd` コマンド `Cmd` を `n` 回繰り返す（例：5w, 20j）

- ファイル操作

  `:eFile` ファイル File を編集
  `:rFile` ファイル File をカーソル位置に読込
  `:w` 保存 (write)
  `:wFile` 別名 File で保存
  `:!Cmd` シェルコマンド Cmd 実行

9.3 削除・移動・コピー

vi では、削除・コピーした文字はバッファに保持される。文字の移動・コピーはこのバッファを経由して行なう。

- テキストの削除

  `x` カーソル位置の文字削除

  `X` カーソル前位置の文字削除

  `dw` カーソル位置のワードを削除

  `d^` カーソル位置の行先頭まで削除

  `d$` カーソル位置から行末尾まで削除

  `dd` カーソル行を削除

  `dfX` カーソル位置から文字 X の出現位置まで削除

- テキストの挿入（バッファからコピー）

  `p` カーソル後ろに挿入

  `P` カーソル前に挿入

- テキストのコピー（バッファへコピー）

  `y^` カーソル位置の行先頭までコピー

  `y$` カーソル位置から行末尾までコピー

  `yy` カーソル行をコピー

  `yfX` カーソル位置から文字 X の出現位置までコピー

9.4 探索と置換

- 文字の書換え

  `r` 1 文字の書換え

  `R` [esc] 入力まで 1 文字ずつ書換え

  `s` 1 文字を [esc] 入力までの文字列に書換え

  `cw` 単語を書換え

  `cc` カーソル行を書換え

  `cfX` 文字 X の出現位置までを書換え

  `u` アンドゥ(取り消し）

  これらも、`10u` のように繰り返しを指定して使える。

- 探索

  `/Pattern` 順方向に Pattern を探索（続けて探索は `n`）

  `?Pattern` 逆方向に Pattern を探索（続けて探索は `n`）

- 置換

  `:s/Pat1/Pat2/` パタン Pat1 を Pat2 に置換え（行内の 1 つ）

  `:s/Pat1/Pat2/g` パタン Pat1 を Pat2 に置換え（行内の全て）

  `:s/Pat1/Pat2/c` パタン Pat1 を Pat2 に置換え（確認付）

  `:1,10s/Pat1/Pat2/` パタン Pat1 を Pat2 に置換え（1-10 行）

  `:1,$s/Pat1/Pat2/g` パタン Pat1 を Pat2 に置換え（1-最終行）

ここでパタンに`/`が含まれる場合は、別の文字を区切りとして使えばよい。またパタンには正規表現が使える。正規表現の詳細は、5 節を参照のこと。

10 emacs エディタ

emacs エディタは、LISP 言語で拡張が可能なエディタであり、極めて多様な機能が実現されている。環境によっては、emacs の代わりに mule、xemacs、meadow など emacs 系列のエディタが用意されることもある。これらも操作は同じである。

10.1 基本的な操作

- 立ちあげ

  `emacs File` File の編集開始

- キー操作の基本

  `C-x` Control キーを押したまま x をタイプ

  `M-x` Escape キーを押し、離してから x をタイプ

- 終了

  `C-x C-c` Emacs を終了

  `C-z` Emacs をサスペンド（`fg` で復帰）

- コマンドの中断

  `C-g` 現在実行中のコマンドを中断

  `C-x u` 最後の編集操作をアンドゥ (undo)

10.2 基本的な編集操作

- ファイル操作

  `C-x C-f` ファイルを読込 (find)

  `C-x i` ファイルを挿入 (insert)

  `C-x C-s` カレントバッファを保存 (save)

  `C-x C-w` カレントバッファを名前指定で保存 (write)

- カーソル移動

  `C-f` 1 文字前進 (forward)

  `C-b` 1 文字後退 (backward)

  `C-p` 手前の行へ (previous)

  `C-n` 次の行へ (next)

  `M-f` 1 ワード前進 (forward)

  `M-b` 1 ワード後退 (backward)

  `C-a` 行頭へ

  `C-e` 行末へ (end)

  `M-a` 1 センテンス後退

  `M-e` 1 センテンス前進

  `M-[` 1 パラグラフ後退

  `M-]` 1 パラグラフ前進

  `C-v` 1 スクリーン前進

  `M-v` 1 スクリーン後退

  `M->` バッファ末尾へ

  `M-<` バッファ先頭へ

  `M-x goto-line` 第 n 行目へ

  `C-l` スクリーン再描画

- コマンドの繰り返し

  `M-Num Command` コマンド Command を Num 回繰り返す

10.3 削除とコピー

emacs では、コピー/移動はキルリングと呼ばれる作業域を経由して行なわれる。まず、対象領域をキルリングへコピー/移動し、これを別の場所へペーストする。対象領域の指定にリージョンがある。リージョンとはマークした位置と現カーソル位置との間のことである。例えばリージョンを使った移動は次のように行なう。移動先頭位置をマークし、移動領域の末端にカーソルを移動してキルコマンドでキルリングに内容を移動し、次に別の場所へカーソルを移動してペーストコマンドを出す。

- リージョンの指定

  `C-SPACE` 現在位置をリージョンの端点としてマーク

  `C-x C-x` カーソルとマークの位置を入れ換え

  `M-h` 現パラグラフをリージョン指定

  `C-x h` バッファ全体をリージョン指定

- テキストの削除/キル

  `DEL` 手前の文字削除

  `C-d` カーソル位置の文字削除 (delete)

  `M-DEL` 手前のワードをキル

M-d　カーソル位置のワードをキル

C-k　カーソル位置から行末までキル (kill)

C-w　リージョン（マーク位置とカーソル位置の間）をキル

M-w　リージョンをキルリングにコピー

- キルしたテキストの挿入

C-y　直前にキルした内容を挿入 (yank)

M-y　以前にキルした内容を挿入（C-y の後に実行）

- レクタングルコマンド

M-x kill-rectangle レクタングルをキル

M-x yank-rectangle レクタングルを挿入

10.4　サーチと置換

- インクリメンタルサーチ

C-s　順方向にサーチ開始/繰り返し (search)

C-r　逆方向にサーチ開始/繰り返し (reverse)

ESC　サーチを抜ける

- 単純サーチ

C-s ESC 順方向にサーチ開始

C-s　順方向にサーチ繰り返し

C-r ESC 逆方向にサーチ開始

C-r　逆方向にサーチ繰り返し

- 対話型置換

M-%　対話型置換の開始

y　置換して次へ

n　置換せず次へ

!　残りを一括置換

ESC　対話型置換の終了

C-r　リカーシブエディット（置換を一時中断したエディット）開始

M-C-c　リカーシブエディット終了し、対話型置換へ戻る

- 検索置換

M-x replace-string 文字列の一括置換

- 正規表現サーチと置換

M-x re-search-forward 順方向の単純な正規表現サーチ

M-x isearch-forward-regexp 順方向のインクリメンタル正規表現サーチ

M-x query-replace-regexp 対話型の正規表現置換

M-x replace-regexp 一括の正規表現置換

- 正規表現演算子

^　行の先頭にマッチ（^abc は abc で始まる行）

$　行の末尾にマッチ（xyz$は xyz で終る行）

.　任意の 1 文字にマッチ

.*　任意の文字列（空を含む）にマッチ

¥<　ワードの先頭にマッチ

¥>　ワードの末尾にマッチ

10.5　バッファとウインドウ

ウインドウは画面上に開いている窓のこと、バッファはその窓を通して操作される編集単位のことである。ウインドウにはバッファの一部が表示される。ウインドウに表示されないバッファも裏で保持されている。編集は emacs が保持するバッファに対してなされ、ディスク上のファイルを直接扱っているのではない。明示的に書き込まなければファイルは更新されない。

- バッファ操作

C-x b 指定バッファへ移動 (buffer)

C-x C-b バッファの一覧表示

C-x k 指定バッファを削除

- ウインドウ操作

C-x 2 カレントウインドウを上下分割

C-x o 次のウインドウに移動

C-x 0 カレントウインドウを削除

C-x 1 カレントウインドウ以外を削除

C-x ^ ウインドウの縦幅拡大

10.6 カナ漢字変換 (Canna)

複数の機能を持つキーは、モードによって機能を変化する。例えば C-n は、キーイン直後では文字種変更、SPACE を押して変換を始めた後では次候補、もう一度 SPACE を押して一覧が出た段階では次の一覧を出す機能を持つ。モードを戻すには DEL キーを押す。

C-¥ アルファベットと日本語入力モード切替え

SPACE 変換/次候補

RETURN 確定

C-g 中止

C-f カーソル右/右候補/右文節選択

C-b カーソル左/左候補/左文節選択

C-p 前文字種/前候補/前候補一覧

C-n 次文字種/次候補/次候補一覧

C-i 文節を短く

C-o 文節を長く

C-a 読みの左端へ/左端候補/左端文節選択

C-e 読みの右端へ/右端候補/右端文節選択

DEL カーソル左文字削除/漢字を読みに戻す

C-k カーソル右文字列削除/部分確定

C-d カーソル文字削除

C-e 編集モードへ

INS 記号入力

C-u 大文字へ

C-l 小文字へ

C-w 候補一覧/部首一覧

C-y 16 進コード変換

C-q クオーテッドインサート

M-x canna-extend-mode 記号、コード、部首入力等

10.7 その他のコマンド

- キーボードマクロ

C-x (マクロ定義開始

C-x) マクロ定義終了

C-x e マクロ実行（定義されたキーシーケンスを再生）

- ヘルプコマンド

C-h b カレントバッファのキーバインドを表示

C-h k 指定キーストロークにバインドされたコマンドを表示

C-h w 指定コマンドをバインドしているキーを表示

C-h l 最後にタイプした１００個のキャラクタ

C-h T emacs のチュートリアル

C-h i Info の実行

- Info 機能

emacs に関連する情報を見ることができる。起動は C-h i。

q 終了

SPACE　1 スクリーン前進

DEL　1 スクリーン後退

m　メニューから指定したノードへ移動

u　上位のノードへ移動

n　次のノードへ移動

p　前のノードへ移動

f　相互参照機能によりノードを移動

l　直前にいたノードへ移動

d　起動画面へ戻る

s　正規表現探索

10.8　C プログラミング便利機能

- インデント

TAB　カーソルのある行をインデント

C-j　次の行へ移動し、インデント

M-C-¥　リージョン内の各行をインデント

M-m　行の最初の非ブランク文字へ移動

M-;　コメント用のインデント

M-x set-c-style インデントスタイルの設定。?で候補の一覧表示

M-x untabify リージョン内のタブコードをスペースに変換

- コンパイル

M-x compile コンパイル実行（最初の実行ではコンパイルコマンドを要求されるので、gcc prog.c -o prog 等と指定。makefile を使う場合はそのままで良い。）

C-x ‘　エラー行に飛ぶ。

- UNIX コマンド実行 (コンパイルに成功した実行ファイルも含む)

M-!　UNIX コマンドを実行

M-|　リージョンを入力として UNIX コマンド実行

M-x shell シェルモードに入る。

- etags

説明 : ファイルとその中に含まれる関数の対応表を作り、それを利用して複数ファイルの間でサーチ/ジャンプを行なう。

準備 : シェル中で etags コマンドを実行し、タグテーブルファイル TAGS を作成する。カレントディレクトリの *.c, *.h について作るには、etags *.c *.h。さらに emacs 中で、M-x visit-tags-table を実行しタグテーブルの位置を指定。

実行 :

M-.　関数名のサーチ開始

M-,　次の候補サーチ

M-x tags-search 正規表現文字列をタグテーブル中のファイル群からサーチ

M-,　次の候補サーチ

- Dired（ディレクトリエディタ）

開始 : ファイル名の代わりにディレクトリ名を指定する。例えばディレクトリ doc は emacs doc。カレントディレクトリは emacs .。

n カーソルを次に移動

p カーソルを前に移動

v ファイル内容表示

e ファイル編集

r ファイル名変更

c ファイルコピー

d 削除マークを付ける

x 削除マーク付きのファイルを削除

11 プログラミング

11.1 シェルプログラミング

複数のコマンドを一つのファイルにまとめておいて実行することが出来る。例えば

File: ar.sh
```
tar cvf report.tar ~/report
tar cvf prog.tar ~/prog
tar cvf data.tar ~/data
```

のようにコマンドの並んだファイルを作っておき、「sh ar.sh」として実行すると各行のコマンドが順に実行する。また先頭に「#!/bin/sh」のような実行環境を指定する行を追加して、「chmod +x ar.sh」でファイルを実行可能に設定すると、「./ar.sh」のようにコマンドとして実行できる。

下のより複雑な例は、「./bak.sh *.c」のようにして、ファイルのバックアップコピー「xx.c ⇒ xx.c.bak」を作るシェルスクリプトである。「*.c」はシェルにより、ディレクトリ中の全ての C ファイルの列に展開されて、スクリプトに渡される。これを全引数を表す$*で受け、一つずつ変数 filename に取り出して処理する。変数 count は、declare -i count で整数に定義している。定義が無い filename のような変数は文字列である。「$」で始まる文字列は、対応する変数の値に置き換えられる。数字 n を使った「$n」は引数の n 番目を表す。

File: bak.sh
```
#!/bin/sh
declare -i count
count=0
for filename in $*
do
  cp $filename $filename.bak
  count=$count+1
done
echo "Convert $count files"
```

ループや条件分岐などを使った記述も可能である。詳細は調べること。「/etc/rc.d/」の下には、システム起動時等に自動実行されるシェルスクリプトが多数存在する。

11.2 Perl

Perl に慣れると、ちょっとしたファイル編集が簡単に行える。以下のようなデータファイル usage.log があるとき、その各行からユーザ名を取り出して表示する Perl スクリプトを getuser.pl に示す。ここで while は標準入力の各行についての繰り返しであり、その次の行は、正規表現で説明したパタンマッチを行うものである。

File: usage.log
```
xxxxxxxxxxx user=01234567, xxxxxx
xxxxx user=01357924, xxxxxxx
```

File: getuser.pl

```
#!/usr/local/bin/perl
while(<>){
  /user=(.*), /;
  print "$1\n";
}
```

このスクリプトは、以下のようにして動かせる。

```
chmod +x getuser.pl
./getuser.pl <usage.log  >userlist
```

更にコマンドを組み合わせて、重複したユーザ名を削除したリストを作ることもできる。ここで、sort はソートコマンドであり、uniq は重複行を削除するコマンドである。

```
./getuser.pl <usage.log | sort | uniq >userlist
```

最近では、Perl の代わりに、Ruby、PHP、Python などの言語も使われる。

11.3 gcc の使い方

C プログラムのソースファイル prog.c をコンパイルする最も単純な方法は

```
gcc prog.c
```

である。この結果、実行ファイル a.out が作られる。通常は、実行ファイルの名前 prog を指定して、

```
gcc prog.c -o prog
```

とする。複数のソースファイル prog1.c,prog2.c,prog3.c から実行ファイル prog を作るには

```
gcc prog1.c prog2.c prog3.c -o prog
```

とする。
gcc には`-o` 以外にも、極めて多くのオプションが用意されているが、ここでは有用と思われる少数のオプションのみを掲げる。詳細は “man gcc” でオンラインマニュアルを見ること。

`-o file` 出力をファイル file に対して行なう。このオプションを省略すると実行ファイルの出力はファイル a.out に対して行なう。

`-g` gdb デバッガのための情報を出力する。

`-ansi` 全ての ANSI 標準の C プログラムをサポートする。このため、GNU C の持つ ANSI C と非互換のいくつかの機能を無効にする。

`-Wall` より多くの警告メッセージを出す。

`-Wmissing-prototypes` プロトタイプ宣言の抜けに警告を出す。

`-l library` リンクに際して、ライブラリ library を使う。（＊数学関数利用時は`-lm` をオプションに加える必要がある）

`-L dir` リンクに際して、ライブラリを探すディレクトリに dir を加える。

`-I dir` 前処理に際して、インクルードファイルを探すディレクトリに dir を加える。

11.4 代表的標準関数

ANSI 規格で定められている C 言語の標準関数の中から利用頻度の高いと思われる関数を示す。

- `<ctype.h>`

  `int isalnum(int c)` c がアルファベットか数字ならば非 0 を返す。
  `int isalpha(int c)` c がアルファベットなら非 0 を返す。
  `int iscntrl(int c)` c が制御文字ならば非 0 を返す。
  `int isdigit(int c)` c が数字ならば非 0 を返す。
  `int isgraph(int c)` c がスペース以外の印字可能文字ならば非 0 を返す。
  `int islower(int c)` c が小文字ならば非 0 を返す。
  `int isprint(int c)` c がスペースを含む印字可能文字ならば非 0 を返す。
  `int ispunct(int c)` c が区切り文字（`'!'～'/','<'～'@','['～'`','{'～'~'`）ならば非 0 を返す。
  `int isspace(int c)` c が空白文字（`' ','\n','\f','\r','\t','\v'`）ならば非 0 を返す。
  `int isupper(int c)` c が大文字ならば非 0 を返す。
  `int isxdigit(int c)` c が 16 進数の文字ならば非 0 を返す。
  `int tolower(int c)` c が大文字ならば対応する小文字に変換して返す。
  `int toupper(int c)` c が小文字ならば対応する大文字に変換して返す。

- `<math.h>`

  `double cos(double x)` $\cos x$ の値を返す。
  `double sin(double x)` $\sin x$ の値を返す。
  `double atan(double x)` $\tan^{-1} x$ の値を返す。
  `double exp(double x)` e^x の値を返す。
  `double log(double x)` $\log_e x$ の値を返す。
  `double log10(double x)` $\log_{10} x$ の値を返す。
  `double pow(double x,double y)` x^y の値を返す。
  `double sqrt(double x)` $\sqrt{x}$ の値を返す。
  `double fabs(double x)` $|x|$ の値を返す。

- `<stdio.h>`

  `int fclose(FILE *stream)` ストリーム `stream` を閉じる。

`FILE *fopen(const char *filename,const char *mode)` stream を開く。

`int fprintf(FILE *stream, const char *format,...)` stream に出力する。

`int fscanf(FILE *stream,const char *format,...)` stream から入力する。可能なら使用を避ける。

`int printf(const char *format,...)` 標準出力 stdout に出力する。

`int scanf(const char *format,...)` 標準入力 stdin から入力する。可能なら使用を避ける。

`int sprintf(char *s, const char *format,...)` 文字列 s に出力する。

`int sscanf(const char *s,const char *format,...)` 文字列 s から入力する。

`int fgetc(FILE *stream)` stream から 1 文字入力し、その文字を返す。

`char *fgets(char *s,int n,FILE *stream)` stream から n-1 文字（または改行まで）を s に入力し、s のポインタを返す。

`int fputc(int c,FILE *stream)` 文字 c を stream に出力する。

`int fputs(const char *s,FILE *stream)` 文字列 s を stream に出力する。

`int getc(FILE *stream)` stream から 1 文字入力し、その文字を返す。

`int getchar(void)` 標準入力 stdin から 1 文字入力し、その文字を返す。

`char *gets(char *s)` 標準入力 stdin から 1 行入力し s にセットする（'\n' が抜かれる）。可能なら使用を避ける。

`int putc(int c,FILE *stream)` 文字 c を stream に出力する。fputc と同一機能であるが、通常マクロで実現される（より高速だが副作用に注意）。

`int putchar(int c)` 文字 c を標準出力 stdout に出力する。

`int puts(const char *s)` 文字列 s を標準出力 stdout に出力する。

- <stdlib.h>

`double atof(const char *nptr)` 文字列 nptr を実数に変換して返す。

`int atoi(const char *nptr)` 文字列 nptr を整数に変換して返す。

`long int atol(const char *nptr)` 文字列 nptr を long 整数に変換して返す。

`int rand(void)` 疑似乱数を発生し、その値を返す。

`void srand(unsigned int seed)` 疑似乱数のシード（種）を与える。

`void *calloc(size_t nmemb,size_t size)` nmemb x size バイトの動的メモリを割り当てる。

`void free(void *ptr)` ptr で示す動的メモリ領域を解放する。

`void *malloc(size_t size)` size バイトの動的メモリを割り当てる。

`void exit(int status)` プログラムを終了する。

`int abs(int j)` 整数 j の絶対値を返す。

`long int labs(long int j)` long 整数 j の絶対値を返す。

- <string.h>

`char *strcpy(char *s1,const char *s2)` 文字列 s2 を別の文字列領域 s1 へコピーする。可能なら使用を避ける。

`char *strncpy(char *s1,const char *s2,size_t n)` 文字列 s2 を別の文字列領域 s1 へ最大 n バイトコピーする。

`char *strcat(char *s1,const char *s2)` 文字列 s2 を別の文字列 s1 の後ろに連結する。可能なら使用を避ける。

`char *strncat(char *s1,const char *s2,size_t n)` 文字列 `s2` を別の文字列 `s1` の後ろに最大 `n` バイト連結する。

`int strcmp(const char *s1,const char *s2)` 文字列 `s1` と `s2` を比較し、一致したら 0 を返す。

`int strncmp(const char *s1,const char *s2,size_t n)` 文字列 `s1` と `s2` を最大 `n` バイト比較し、一致したら 0 を返す。

`char *strchr(const void *s,int c)` 文字 `c` を文字列 `s` の中から探し、あればその文字へのポインタを、なければ `NULL` を返す。

`char *strrchr(const char *s,int c)` 文字 `c` を文字列 `s` の中から逆向きに探し、あればその文字へのポインタを、なければ `NULL` を返す。

11.5 make の使い方

make はコンパイル・インストールなどの処理を自動化するツールである。ファイル Makefile に依存関係と作成方法を記述しておけば、「`make`」と入力することにより、依存関係に沿って自動処理を行う。

例えば、2 つの C ファイル（main.c、iofunc.c）と、その 2 つに共通のヘッダファイル（myheader.h）から、コマンド (mycmd) を作成する場合を考える。この場合、コマンド mycmd は、オブジェクトファイル（main.o iofunc.o) から、オブジェクトファイル main.o は main.c と myheader.h から、同じくオブジェクトファイル iofunc.o は iofunc.c と myheader.h から作られる。以下はこの依存関係を記述した Makefile の例である。

File: Makefile

```
mycmd: main.o iofunc.o
    gcc main.o iofunc.o -o mycmd

main.o: main.c myheader.h
    gcc -c main.c

iofunc.o: iofunc.c myheader.h
    gcc -c iofunc.c

clean:
    rm -f mycmd main.o iofunc.o
```

構文を下に示す。ここで target は作成するもの、prerequisites は事前に作成が必要なものを示す。target の作成方法は、TAB コードから始まる次行以降に記述する。make コマンドは、依存関係を辿り target を作成する。target が最新状態にある場合には作成をスキップする。

Makefile Format

```
target: [prerequisites ...]
[TAB] command1
[TAB] commnad2
 ...
```

上の例の場合、「make mycmd」と指示すれば、「mycmd：」の記述に従って作成作業を開始する（単に「make」としても良い。最初の target が指定されたとみなされる）。なお、Makefile には作成だけでなく、「clean:」の例のように削除作業なども記述できる。
Makefile には様々な簡易記述法が用意されている。詳細は調べること。

11.6 gdb の使い方

gdb は gcc コンパイラに対応するデバッガである。gdb を使うには、デバッグ情報を付加するために、gcc にオプション「-g」を加えてコンパイルしなおす。

```
gcc prog.c -o prog -g
```

作成された実行ファイルを gdb 上で実行しながら処理状況を見る。

```
$ gdb prog              実行ファイル prog について gdb 開始
(gdb) break 1           1 行目にブレークポイント設定
(gbd) run arg1 arg2     引数 arg1,arg2 付きで実行開始、ブレークポイント停止
(gbd) next              次の行へ進む
(gbd) step              １ステップ進む
(gbd) print var1        変数 var1 の値を見る
(gbd) continue          続けて実行する
(gbd) frame             関数呼び出し関係を表示
(gbd) up                必要なら呼び出し元へ移動
(gbd) list              停止位置のプログラムリストを表示
(gbd) quit              終了
```

適当な行にブレークポイントを設定し、そこまで動かしてから、少しずつ実行しながら変数値と行の変化を追う。next では関数も 1 行として次の行の開始位置まで実行する。step は関数呼び出しで関数内に移動する。continue は次のブレークポイントまで実行を継続する。また、停止時には print で変数値を、frame で関数呼び出し状況を, list でソースリストを見ることも出来る。
gdb は、異常終了時にできる core ファイルを使ってデバッグすることもできる（core ファイルのサイズがゼロのときは ulimit コマンドを参照のこと)。

```
./prog arg1 arg2
セグメントエラー (core を出力しました)
```

`gdb prog core.12345`	実行ファイル `prog` と `core` ファイル `core.12345` で開始
`(gdb) frame`	関数呼び出し関係を表示
`(gdb) up`	必要なら呼び出し元へ移動
`(gbd) list`	停止位置のプログラムリストを表示
`(gbd) print var1`	変数 `var1` の値を見る
`(gdb) quit`	終了

ここではコマンドは分かりやすく表記したが、省略形が使えるので多くの場合は最初の 1-2 文字だけ入力すれば動く。また TAB 補完もできる。その他、多様なコマンドがある。詳細は help を参照のこと。

参考文献

[1] Debian リファレンス, http://www.debian.org/doc/user-manuals#quick-reference

[2] アットマーク・アイティ Linux Square, http://www.atmarkit.co.jp/flinux/