OUKA
The University of Osaka
Institutional Knowledge Archive

Title	UNIXにおけるプログラム開発環境
Author(s)	山口, 英
Citation	大阪大学大型計算機センターニュース. 1989, 72, p. 23-40
Version Type	VoR
URL	https://hdl.handle.net/11094/65817
rights	
Note	

The University of Osaka Institutional Knowledge Archive : OUKA

https://ir.library.osaka-u.ac.jp/

The University of Osaka

UNIXにおけるプログラム開発環境

大阪大学基礎工学部情報工学科 山口 英[†]

はじめに

UNIX[1]はこれまで特にプログラム開発用の OS として注目されてきたシステムであり、実際の商業的なソフトウェアの開発や研究目的でのソフトウェア開発などに非常に多く使われている。これは、現在の UNIX では多くの言語がサポートされソフトウェアの目的にあった言語を比較的自由に使用できることや、ソフトウェア開発に使用できる各種のツール・ライブラリが用意されていることなどが理由として挙げられる。また、UNIX はそれ自体が C 言語で開発されたことから C 言語によるソフトウェア開発の環境が良く整備されていることからも注目されている。

この解説では UNIX[2]のプログラム開発環境、特に C 言語によるソフトウェア開発環境について、そのためのツールと効果的な利用方法の解説を通して説明する[3]。

C言語によるソフトウェア開発の概略

UNIX における典型的な C 言語によるソフトウェア開発の段階を図 1に示す。

まず、デザインにしたがってソースファイルを作成する。これには一般にエディタが用いられるが、UNIX では非常に簡単なものから高機能で強力なものまで、さまざまなエディタが用意されている。UNIX に標準で用意されているものとしては、ed[2],ex[3], vi[4]がある。これ以外に利用できるエディタとしては、非常に強力で拡張性に富んだ GNUEmacs[5,6], UNIPRESS Emacs, jove[7]等が挙げられる。

次にソースファイルをコンパイル・リンクして実行形式ファイルを得る。これにはコンパイラである cc を利用する。コンパイルの実行を管理する make[8]というコマンドや、C 言語の文法チェックだけを行う lint[9]というコマンドも用意されている。

実行形式ファイルが得られたら、それを実際に実行させて動作チェックを行う。動作チェックで、問題がある場合はソースファイルの変更やデバッガを利用してデバッグを行う。UNIX にはデバッガとして adb[10], dbx[11]がある。

当初の目的が達成されたら、作成したソフトウェアをリリースする。この後で大抵の場合(特に商業目的のソフトウェア作成の場合)ソースファイルの管理をしなければならない。UNIX ではソースファイルの管理においても rcs[12], sccs[13]などのツールを提供している。この後、新しい機能の追加や、リリース後のバグの修正などによって、このサイクルを続けることになるのが普通である。

このように、UNIX ではソフトウェアの開発・保守までを含めて非常に多くのツールが(大抵の場合標準で)提供されている。

エディタ

ソフトウェア開発をする場合、始めにソースファイル作成を行わなければならない。この場合エディタを使用するが[4]、効率の良いソースファイル作成をするには使用するエディタの機能を十分に利用できるかどうかが鍵になる。この節では UNIX に用意されているエディタの紹介と便利

[†]yamaguti@osaka-u.junet

[1]UNIX は AT&T ベル研究所が開発し、AT&T がライセンスしているオペレーティングシステム。

[2]この解説では 4.[23]Berkeley UNIX を指す。EWS4800等の System V 系の UNIX でも同様の環境が提供されているが、Berkeley 系の UNIX の方が多くの PDS のツールが利用可能であり、安価で効果的な環境が作りやすい(大学・研究機関向き)。

[3]UNIX における簡単な C プログラムの作り方についてはセンターニュース前号[1]においてされているので参考にしてもらいたい。

[4]真のハッカーはソースファイル作成に cat とシェルリダイレクションを使うという joke があった。

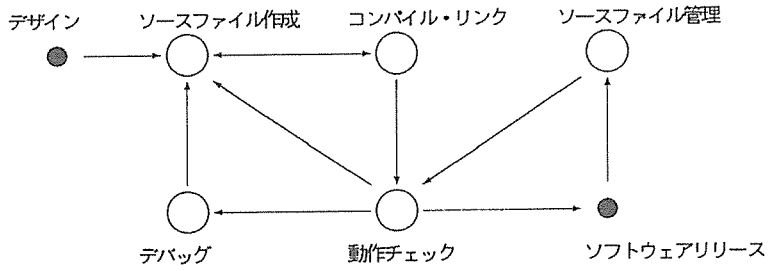

図 1: UNIX における C 言語によるソフトウェア開発

な使い方について簡単に説明する。

どんなエディタが利用可能か

UNIX では標準で ed, ex, vi が用意されている。ed は UNIX で用意されているもっとも簡単なラインエディタであり、ex, vi などの基礎になっているが、これをユーザが直接使ってソースファイルを作成することは現在ではあまりされていない[5]。一方、ex は ed を拡張したラインエディタであり、ed と比べてかなり高機能になっている。さらに、この ex をスクリーンエディタに拡張したものが vi である。したがって、これら 3 つのエディタではコマンドや約束には、似ている点がかなりある。

標準で用意されている以外のエディタでは Emacs が有名である。Emacs は CMU の James Gosling によって開発されたエディタである。現在では UNIX 上では、GNU Emacs が多くの計算機で使われている。この GNU Emacs には NEmacs と呼ばれる日本語化されたバージョンがある。また、UCB で開発された jove というエディタは、そのインタフェースや機能が Emacs と同じで、現在では 4.3BSD のディストリビューションテープに入れられている。PDS では Micro Emacs があり、これは日本語化されて kemacs として配布されている。Micro Emacs はその名の通り、Emacs の幾つかの機能を提供しているエディタであるが、UNIX 以外にもパーソナルコンピュータ(例えば PC9801)などでも動作するという利点があり、利用者は多い。

[5]計算機管理者の場合は、シングルユーザモードの時に作業をすること(例えば OS のバージョンアップの時)がしばしばあり、多くの場合 ed 以外のエディタが使えない状態で作業をしなければならないので、ed を使えるようなっていることが必要とされる。

正規表現

UNIX 上のエディタを効果的に使うコツは、正規表現になれることである。正規表現というのは UNIX 全体に共通する考え方で、検索や置換の場合に文字列のパターンを指定するのに使われ、この表現を使った検索・置換の機能をほとんどのエディタが提供している。正規表現を用いると複雑なパターンや文字列の構造が簡単に記述でき、さまざまな検索・置換をすることが可能になる。表 1に正規表現の規則を示す。これらの正規表現にマッチする文字列が 1 行にたくさんある場合は、その中でもっとも左側にあるものにマッチする。次にこの正規表現を使った例を示す。

`abc`	abc という文字列。
`[a-zA-Z]*`	任意の長さ(0 を含む)の英字列。
`[A-Z][a-zA-Z]*`	大文字から始まる英字列。
`\^.\$`	1 文字から構成される行。
`\([Aa]bc\)\1`	AbcAbc か abcabc にマッチする。`\([Aa]bc\)` は Abc か abc を表し `\1` はそのどちらかにマッチする。

タグ付きの正規表現はエディタにおいて置換を行う時に役に立つ。例えば ex の置換コマンドで s/α/β/ は、現在の行の α という文字列を探し、β に置き換える。次のように指定すると、文字列“Yamaguchi Suguru”を“Suguru Yamaguchi”

c	特別な意味を持たない文字 *c* 、それ自身を表す
c	文字 *c* の特別な意味をなくす
^	行の先頭
$	行の終り
.	任意の文字 1 文字
[...]	...の内の任意の 1 文字, a-z のような形式で範囲の指定も可能 順序は ASCII コードによって決められる
[^...]	...に含まれない任意の文字 1 文字
\n	*n* 番目の\(...\)がマッチしたものを指す
*r**	正規表現 *r* の 0 回以上の繰り返し
r1r2	正規表現 *r1r2* の並び
\(r\)	タグ付き正規表現 *r* (ネスト可能)

表 1: 正規表現(優先度の低い順)

に置き換えることができる。

```
s/\(Yamaguchi\) \(Suguru\)/\2 \1/
```

ed の便利な機能

UNIX でもっとも基本的なエディタである ed は非常に簡単な機能だけを提供しているが、複数のファイルに対して同時に同じ変更を加える場合などによく使われる。編集のためのコマンドファイルを *comfile* とすると、

```
ed datafile < comfile
```

とやると datafile が変更できるので、これをシェルプログラムなどで繰り返し実行すると効率の良い編集ができる。

また ed はファイルの差分保存にも使うことができる。2 つのファイルの内容を比較し、その差分を出力する *diff* というコマンドでは、

```
diff -e file1 file2
```

とすると、*file1* から *file2* を作る ed コマンドを出力する[6]。これを利用すると、同じようなファイルの幾つも保存する場合や、プログラムの履歴を残す場合に非常に便利である。

[6]出力されるコマンド列には出力コマンドが含まれていないことに注意。この出力をファイルに保存し ed に入力し変更するには出力を保存したファイルの最後の行に wq を付け加える必要がある。

スクリーンエディタ vi

UNIX で一般的に使われるエディタは vi である。これはラインエディタである ex をスクリーンエディタに拡張したもので、現在の UNIX でもっとも良く使われているエディタである。これをうまく使いこなすには初期設定、ex コマンド、名前つきバッファ等を効果的に使うことが重要である。この節では vi (ex)の便利な使い方について簡単に紹介する。

初期設定

vi(ex)の初期設定をファイル".exrc"に書いておくか、環境変数 EXINIT に同様の設定をすることで行われる[7]。カレントディレクトリの".exrc"の設定がもっとも優先され、それが無い場合は環境変数 EXINIT での設定、ホームディレクトリの".exrc"の設定の順で優先される。

初期設定ファイルでは幾つかのオプションが設定できる。例えば、表示で行番号をつけるようにするには、set コマンドを使って

```
set number
```

とし、この設定を解除する場合は、`set nonumber`

[7]`setenv EXINIT "set ai redraw nonu"`というようにして設定される。

オプション	短縮名	機能	default
`autoindent`	ai	改行した時に、その前の行と同じだけ段付けをしてくれる。もしも段付けを一つ浅くする場合は、そこでCTRL-d を打てばよい。段付けを深くする場合はCTRL-t かタブを入力する。	noai
`number`	nu	行番号を表示する。	nonu
`autowrite`	aw	編集中にファイルの切替えが":n"や":ta"で発生した場合や、編集がCTRL-z や":su"でなされた場合に自動的にファイルへの書き込みを行う。同時に複数のファイルを連続して編集する場合は便利である。	noaw
`ignorecase`	ic	検索の場合の大文字と小文字の区別をしていする。noic とすると大文字と小文字を区別する。	noic
`lisp`		lisp のS 式に対応した段付けを自動的に行う。	nolisp
`list`		ファイルの内容で行の終りを"$"で示し、タブを"^ I"で表示する。タブと空白のどちらを使用しているかをチェックする時に便利。	nolist
`sh=`*path*	sh	"!!"やshell コマンドで使用されるシェルのフルパス名をしていする。	sh=/bin/csh
`showmatch`	sm	"{"を入力した時に対応する"}"を探して、一瞬(1 秒)カーソルを対応する場所に移動しすることで、括弧の対応関係のチェックの手助けになる。	nosm
`redraw`		機能の低い端末で高機能端末の動作をvi がシミュレートする。高速で接続されている場合以外はやめた方がよい(端末に送られる画面制御シーケンスが非常に多くなるため)。	noredraw
`tabstop=`*n*	ts	タブストップを *n* に設定する。段付けが深くなるような大きなC プログラムを書く場合、非常に便利。	ts=8
`taglength=`*n*	tl	タグ機能でどれだけの文字を比較対象とするかの文字数を設定する。0 の場合、文字列全体が比較対象となる。	tl=0
`wrapscan`	ws	ws では検索がファイル末に達すると自動的にファイルの先頭に検索が移動する。nows とするとファイル末で検索を中止する。	ws
`wrapmargin=`*n*	wm	入力モードで自動折り返しする場合の一行の幅を *n* に設定する。0 の場合は自動折り返しの機能は使われない。	wm=0

表 2: v i のオプション(一部)

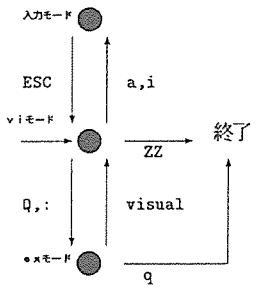

図 2: ｖｉでのモード

というようにオプション名の前に“no”をつけて set コマンドを実行する。表 2に便利なオプションを示す。

初期設定ファイルでは、オプション以外に略記 (abbreviation) の機能がある。これはコマンド ab で

```
ab #i #include
```

と設定しておくと、“#i”の入力後に空白などの区切り記号が入力されると“#include”に展開される。この機能を用いて頻繁に使う言葉について設定しておけば、入力が非常に楽になる。しかしながら、略記をうまく設定しないと不要な部分で展開が行われることになるので、注意して使わなければならない。

モード

vi には 3 つのモードがある。一つはコマンドモードで、このモードではカーソル移動などの vi のコマンドが有効である。インサートモードでは“i”や“a”のコマンドを実行した時に ESC を入力するまでのモードである。このモードでは一部のもの (例えば CTRL-v) を除いて vi コマンドは有効ではない。3 つ目のモードは ex モードで、“:”の入力以後、改行が入力されるまでか、“Q”で ex の状態に移った場合であり、この状態では ex コマンド(前節の set コマンドや ab コマンドは ex コマンドである)が有効である。この 3 つのモードの状態遷移を図 2に示す。

vi コマンドと ex コマンドの使い分けが vi をうまく使うためのコツである。例えば、ある 1 箇所だけの文字列の書き換えならば vi モードで“s”コマンドを使って書き換えればいいが、ファイル全体に対して散らばっている同じ文字列を書き換える場合は ex コマンドで“:gs/*str1*/*str2*/g”とやるほうが効率が良い。

便利なｖｉコマンド

vi コマンドにはたくさんの便利な機能がある。便利なコマンドについて表 3にまとめておく。

便利なｅｘコマンド

ある特定の文字列をファイル全体に渡って置き換えたいというような、ラインエディタが得意とするような操作の場合は、ex コマンドが非常に便利である。また、複数のファイルを同時に編集する場合も編集するファイルの切替え等で ex コマンドを使う。

複数のファイルを同時に編集するには vi を起動する時の引数として編集するファイルを複数指定すればよい。例えば、

```
% vi main.c sub.c util.c
```

とした場合、最初に編集するファイルは main.c となる。sub.c を編集したい場合は、main.c に変更があれば一旦ファイルに書き込み以降 (“:w”) 、次にファイルの移動を“:n”コマンドで行う。これを繰り返し、util.c を編集した後に main.c を再度編集したい場合はリワインドコマンド (“:rew”) を使う。これによって引数に対するポインタが戻され、main.c に編集ファイルが移動する。この次に“:n”をすると sub.c に移動する。

また、何らかのシステムのトラブルによって編集中の状態で vi のプロセスが消された場合は、編集状態を残したまま途中終了になる[8]。次に vi を起動した時に“:recover”で編集途中の状態に復元することができる。

そのほか、“.exrc”で設定しているオプションの変更 (“:set”) や略記(“:ab”) の追加なども ex コマンドとして実行できる。ネットワークを介して計算機を使用している場合、端末名の設定がうまくいっていない状態になることがしばし

[8]この場合、ユーザに対してメイルで編集状態が保存されているかどうかを伝えてくる。

コマンド	動作
CTRL-v	入力モードの時、この次に打った文字をそのまま入力する。例えば"CTRL-V ESC"とするとESCがそのまま入力される。コントロール文字を入力する場合に便利。
%	対応する括弧にカーソルを移動させる。
>>	現在カーソルのある行の段付けを一段深くする。
<<	現在カーソルのある行の段付けを一段浅くする。
!!	次に改行を打つまでの文字列をUNIXのコマンドとして実行し、コマンドの標準出力をその行と置き換える。このコマンドを実行する前にカーソルのあった行の内容は無くなるので注意が必要。
(ピリオド)	直前に実行した変更コマンドを繰り返す。

表3: 便利なｖｉコマンド

ばある。例えば、vt100の端末を使っている時に、端末名の設定を間違っているような場合は、viは正しく動作しない。viでは端末名をオプションのttytypeに覚えているが、これを書き換えようとして、

```
:set ttytype=vt100
```

というようにしても設定を変えることはできない。この場合、一旦exに"Q"コマンドで移り、そこでsetコマンドを使って端末名を正しく設定し直して、"visual"コマンドでviに戻るということをすれば、正しく動作するようになる。

名前付きバッファの活用

viには名前無しバッファと名前付きのバッファが用意されており、これを積極的に使うことでカット・ペーストを効果的に行うことができる。

削除コマンドの"dd"や"x"を実行すると削除された行や文字は一旦名前なしバッファに保存される。また、"yy"コマンドを実行すると現在カーソルのある行が名前無しバッファにコピーされる。これを現在のカーソル位置の次の行やカラムに挿入するには"p"コマンドを実行すればよい。

名前付きバッファは英小文字1文字の名前が付けられている。したがってviで用意されている名前付きバッファは26個あることになる。aという名前のバッファは"aで表される。例えば、

```
"add
```

とすると、現在カーソルのある行が削除されてバッファaに保存される。カーソル移動後、これをカーソルの次の行に挿入するには、

```
"ap
```

とすればよい。

ｕｎｄｏ機能

行や文字を削除したり、変更した場合に取消をしたい場合がある。viでは実行したコマンドの取消(undo)の機能を用意している。コマンドを実行した後で、それを取り消したい場合は"u"コマンドを入力すると直前に実行されたコマンドを取り消してくれる。一方、連続して"x"コマンドを複数個の文字を削除した場合、"u"コマンドでは最後に削除した文字だけを復活してくれる。同じ行に対して連続して変更が加えられた場合、その変更をすべて取り消したい場合には"U"コマンドを使えばよい。"U"コマンドは現在行の連続した変更をすべて無効にし、前の状態に戻してくれる[9]。

変更をした直後に取消を実行すれば直ちに復活可能であ

[9]ただし、"dd"コマンドで行を削除した場合の復活は"U"コマンドでは行ってくれない。この場合はやはり"u"コマンドを使うしかない。

る。間違いに気が付くのが遅く、すでに他の行に移動して変更をしていた場合も気が付くのが早ければ復活可能である。vi は9個の番号付きバッファを持っており、これに最近削除したものを最大9つまで保存しておく。このバッファも名前付きバッファと同じように参照する。例えば、3番のバッファに入れられている内容を現在カーソル位置の次の行に挿入するには、"3p とすればよい。バッファに入れられているものが分からない場合は、

```
"1p.u.u.u.u.
```

というように"u"と"."を繰り返すことによって順番にバッファの内容が挿入されていくので、希望のものが得られたら、それを移動させれば良い。"."コマンドは通常は直前に実行したコマンドを再度実行するのに使われるが、この場合はバッファの名前である番号を一つづつ増やしてコマンドを際実行する。

マーク機能

マークは現在を位置を覚えておくための目印で、英小文字一字がマークとして利用できる。マークを付けるには vi コマンドの"m"を使うか、ex コマンドの"k"を使う。例えば、現在行に b というマークを付けるにはコマンドモードで、

```
mb
```

と打てばよい。このマークを使う場合に、マークのある位置(行とカラム)そのものを表す場合と、マークのある行を表す場合で参照の仕方が異なる。前者は`a(逆シングルクオートと a)で参照され、後者は'a(シングルクオートと a)で参照される。例えば、コマンドモードで`a と入力すると、マーク a のある位置にカーソルが移動するが、'a と入力した場合にはマーク a のある行の先頭から見て最初の空白文字でない文字のところにカーソルが移動する。このマークは ex モードでも使え、マーク a からマーク b の間の行における置換は、例えば次のようにすることができる。

```
:'a,'bs/Orange/Apple/
```

タグ機能

C 言語によってソフトウェア開発をする場合、通常ソースファイルはその機能毎に分割されることが多い。この場合、あるファイルを編集していて、その中に使われている関数の宣言がされているファイルを編集したい場合が多い。例えば、ファイル sub1.c で使われている getattr()という関数がどのように宣言されているか知りたい、あるいは、その関数を書き換えたいといった場合を考える。この場合、ソースファイル数やサイズが小さい場合はどこにどの関数が宣言されているかを覚えておくのは可能であるが、ファイル数などが増えてくると、すべてを覚えているのは困難になってくる。このような時に便利なのがタグ機能である。タグ機能を使うと、簡単な操作によって vi は指定した関数が定義されているファイルへ移動する。

タグ機能を使うためには、最初にどのファイルにどの関数が定義されているかというデータベースファイル tags を作らなければならない。これには ctags というコマンドを使う。

```
% ctags *.h *.c
```

とすると、カレントディレクトリの C ソースファイルすべての中で宣言されている関数の情報をファイル tags に生成してくれる[10]。このファイルを作ったのちに、vi の中で getattr の先頭にカーソルを持っていき、CTRL-] を入力すると、getattr()が宣言されているファイルに移動することができる。もしも、現在のファイルに変更があり、ファイルに書き出していない場合は移動しない[11]。ex の場合は、

```
:ta getattr
```

とすると、関数 getattr()の宣言がなされている部分を表示してくれる。ctags は C 言語ソースファイルにしか使うことができないが、tags ファイルの構造は極めて簡単であり、他の言語のソースファイルにタグ機能を使用する場合は自分で tags を生成すればよい。

ソフトウェア開発ツール

ソースファイルを作成した後は、コンパイル・リンクを行って実行形式ファイルを作成し目的を満足するソフトウェアにする作業を行う。UNIX にはコンパイラ、デバッガだけではなく、その他の多くの開発支援ツールが用意されている。UNIX では C によるプログラム開発が中心であるので C 言語を意識したツールが多い。この章ではコンパイラ、

[10] 後述の Makefile にこのコマンドを実行するエントリを付けておくと非常に便利である。

[11] オプションで autowrite が設定している場合には、ファイルに編集結果を書きだし、現在の編集ファイルを移動する。

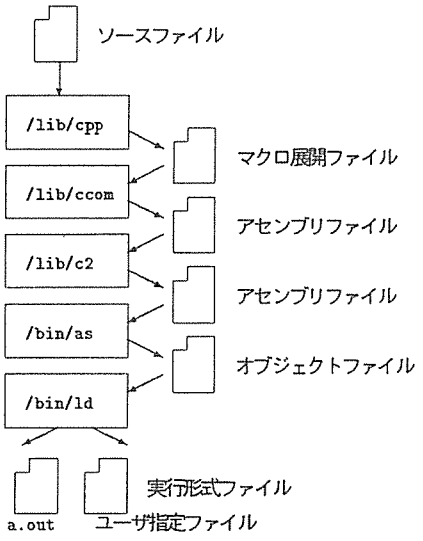

図 3: C コンパイラの処理

デバッガの解説とその他の開発支援ツールの紹介をする。

C コンパイラ

UNIX の C コンパイラとして cc が用意されている。cc はコンパイルからリンクまでに必要なコマンドの起動を自動的に行ってくれるプログラムである(図 3)。

cc を起動すると、オプション解析の後プリプロセッサ /lib/cpp が起動される。/lib/cpp は#include (ファイルの読み込み)、#define (マクロの定義)などのプリプロセッサコマンドの解析を行う。/lib/cpp の実行の後には、#include で指定したファイルの内容が読み込まれ、#define で定義したマクロの展開が行われる。オプションとして -E オプションが指定された場合は、プリプロセッサの実行結果を標準出力に出力し終了する。

次にプリプロセッサを実行した後のファイルが/lib/ccom に渡される。/lib/ccom が本当の C コンパイラであり、実行後アセンブリファイルが出力される。オプションとして -S オプションが指定された場合はアセンブリリストをソースファイル名の".c"をとって".s"にしたファイルに出力し cc は終了する。cc 起動時にオプティマイザの使用を指定した場合(-O オプションを指定した場合)には、この後に/lib/c2 というオプティマイザが起動される。この後にアセンブラ/bin/as が起動されオブジェクトファイルが作成される。オプションとして -c オプションが指定された場合は、オブジェクトをソースファイル名の".c"をとって".o"にしたファイルに出力し実行を終了する。

最後にオブジェクトファイルはリンケージエディタ ld によってライブラリなどとリンクされ、実行形式ファイルが作られる。普通は"a.out"というファイルに実行形式が出力されるが、-o オプションによってファイル名が与えられた場合には、出力される実行形式ファイルはそこに指定されたファイル名となる。

ld は実行形式を作る際に幾つかのライブラリをリンクする。どの C プログラムも/lib/libc.a がリンクされるが、これ以外のライブラリをリンクする際は -l オプションを用いる。UNIX におけるライブラリは

lib ライブラリ名.a

というように統一的な名前の付け方がされている。例えば、数値計算用のライブラリは/usr/lib/libm.a という名前が付けられている。これをリンクするには

```
% cc prog.c -lm
```

という形式で指定すればよい。リンクされるライブラリは標準で/lib, /usr/lib, /usr/local/lib の順に探されていく。

C プリプロセッサ

C プログラムをうまく書く方法として C プリプロセッサの機能を使うことが挙げられる。C プリプロセッサの機能を表 4に示す。

cc には -D オプションと -U オプションが用意されており、プリプロセッサコマンドの#define と#undef に対応する。次にその例を示す。

`#define` *identifier tokenstr*	この宣言以降、プリプロセッサは識別子 *identifier* が現れたならば、それを *tokenstr* で指定された文字列に置き換える。*tokenstr* の指定が長くなった場合はバックスラッシュで次の行に継続することができる。
`#define` *identifier*	この宣言以降、*identifier* が定義されているとする。
`#undef` *identifier*	この宣言以降、*identifier* に対する定義を無効にする。
`#include "`*filename*`"`	*filename* で指定されたファイルの内容全体と置換される。指定されたファイルは、ソースファイルのあるディレクトリで探され、次に一連の標準の場所が探される。
`#include <`*filename*`>`	"*filename*"と同じであるが、ファイルの探索は標準の場所だけで行われ、ソースファイルのあるディレクトリでは行われない。
`#if` *exp*	いわゆる if 文と同じで、*exp* が 0 かどうかのチェックが行われる。0 でない場合には then 節が、0 の場合は else 節が有効となる。
`#ifdef` *identifier*	*identifier* が現在定義されているかどうかを調べる。
`#ifndef` *identifier*	*identifier* が現在未定義かどうかを調べる。
`#if !defined(`*identifier*`)`	`#ifndef` *identifier* と同じ機能をする。!は否定を表す。従って`#if defined(`*identifier*`)`とすると`#ifdef` *identifier* と同じ機能をすることになる。
`#else`	
`#endif`	
`#line` *constant "filename"*	これに続く行を、ファイル *file* の *constant* 行目とみなすようにコンパイラに指示する。

表 4: Cプリプロセッサコマンド

```
-DDEBUG              #define DEBUG
-DTBLSIZE=10         #define TBLSIZE 10
-DLIB=\"/usr/lib/\"  #define LIB "/usr/lib/"
-UOLD                #undef OLD
```

デバッグを行っている時にはしばしば printf を利用してデバッグメッセージを出力することをするが、メッセージを出力する部分を

```
#ifdef DEBUG
fprintf(stderr,"DEBUG:line=%d\n",i);
#endif
```

のようにしておくと、デバッグをしている間は`-DDEBUG`オプションをつけてコンパイルしておけばメッセージが出力され、最終的にはオプションをつけずにコンパイルすればメッセージの出力が抑制される。このようにしておけばソースファイルを変更しなくてもよく、また、再度デバッグの必要性が出てきた時にはコンパイルを再び行うだけでデバッグメッセージを出力するようにすることができる。

また、Berkely 系 UNIX でのライブラリ関数の index()は System V 系の UNIX では strchr()になるので、これを一つのソースファイル両方でコンパイルできるようにするのに、

```
#ifdef SVR1
#define index strchr
#endif
```

といった宣言をすることもしばしば行われる。

ライブラリ

UNIX では各種のライブラリが用意されているが、ar コマンドを使うことで自分用のライブラリを作ることができる。UNIX ではライブラリはオブジェクトファイルを内容としたアーカイブファイルとして作られている。アーカイブファイルというのは幾つかの UNIX のファイルをひとまとまりにしたファイルのことである。これを操作するコマンドが ar である。ar によって新たなファイルをアーカイブに加えたり、削除したり、格納されているファイルの一覧を見たりすることができる。たとえば、/usr/lib/libm.a というファイルは数値計算のための関数のライブラリであるが、

```
% ar tv /usr/lib/libm.a
```

とすると、中に含まれているオブジェクトファイルの一覧が出力される。

さて、自分用のライブラリを作る場合、ar を使って必要なオブジェクトファイルをアーカイブファイルにまとめればよい。例えば、libmy.a をカレントディレクトリのすべてのオブジェクトファイルから作るには

```
% ar cr libmy.a *.o
```

とする。さらに、これをコンパイラからライブラリとして扱うためにはランダム・ライブラリという形式にしなければならない[12]。これには ranlib というコマンドを使い、

```
% ranlib libmy.a
```

とする[13]。

m a k e

大きな C プログラムでは機能毎に関数群をまとめるためにソースファイルの分割がしばしば行われる。例えば、すべてのソースファイルにインクルード(include)されるヘッダファイル inc.h とソースファイル main.c, sub1.c, sub2.c から実行形式 cmd を作る場合を考える。この時に各ファイルのオブジェクトファイルを作成しながらコンパイルしていく場合、次のようなコマンドを順次実行することになる。

```
% cc -c main.c
% cc -c sub1.c
% cc -c sub2.c
% cc main.o sub1.o sub2.o -o cmd
```

ソースファイルの 1 つに変更を加えた場合は、そのファイルのコンパイルとリンクだけをすればよいが、インクルードファイル inc.h を変更した場合はすべてのソースファイルのコンパイルを再度行う必要がある。このようにあるファイルの変更によってどのような再コンパイルをすればよいか、どのファイルが変更されているかといったことを利用者が管理することは、ソースファイルの数の増加に伴って非常に困難になってくる。実行形式を得るために必要なコマンドの起動をシェルプログラムとして記述し、それを実行すれば必ず正しく実行形式ファイルを得ることができるが、毎回すべてのファイルがコンパイルされるために時間がかかってしまう。このような問題を一気に解決してくれるのが make である。

make は Makefile というファイルに、「作られるもの(目的物)とそれを作るために必要なもの(必要物)の関係」と「目的物を得るためのコマンド」を記述し、これを基にして目的物を得るために必要なコマンドを順次実行する。前述の例の場合の Makefile は次のようになる。

```
cmd: main.o sub1.o sub2.o
        cc main.o sub1.o sub2.o -o cmd
main.o: main.c inc.h
        cc -c main.c
sub1.o: sub1.c inc.h
        cc -c sub1.c
sub2.o: sub2.c inc.h
        cc -c sub2.c
```

この例でも分かるように、

目的物: 必要物
　　実行コマンド

という系列の並びになっており、最終的な目的物を得るための処理がトップダウンに記述されている。また、実行コマンドの書かれている行の先頭は必ずタブでなくてはいけないことを注意しなければならない。このような Makefile

[12] System V 系の UNIX では ar で作られたファイルを直接コンパイラが扱うことができるために ranlib というコマンドは無い。
[13] この操作は libmy.a が変更される毎に行わなければならない。

を main.c などのソースファイルのあるディレクトリに用意し、make を実行すれば最終目的物である cmd を作るために必要な処理をファイルの変更された時刻を基にして調べ[14]、コマンドが実行される。このときどのようなコマンドが実行されたかが順次表示される。

make の実行は、実行されるコマンドの終了状態(exit コード)が 0 以外の値をとった時(例えばコンパイルエラーが発生した場合)に中止される。オプションに - k オプションを指定した場合は、これを無視して引続き実行を行う。また、Makefile の実行コマンドの行の先頭に「-」が付けられた場合も同様である。一方、実行コマンドの行の先頭に「@」が付けられた場合は、実行コマンドの表示が抑制される。

実行コマンドを指定する行では次のようなマクロを使うことができる。

$@	目的物のファイル名
$*	目的物のファイル名から".c"などを取ったもの
$<	必要物すべてのリスト
$?	必要物の内、目的物よりも変更された時刻が新しいもののリスト

例えば、変更されたファイルだけをプリントアウトするための Makefile のエントリをこのマクロを使って書くと、次のようにすることができる。

```
print: inc.h main.c sub1.c sub2.c
        pr $? | lpr
        touch print
```

Makefile の中では変数を使うことができる。これはシェル変数と同じように使われ *identifier* = *value* という形式で設定され、*$identifier* という形式で参照される。この変数の設定は Makefile の最初で行わなければならない。例えば、現在のディレクトリでの作業ファイルをすべて消すようなエントリは次のように書くことができる。

```
TARGET = cmd
OBJ = main.o sub1.o sub2.o
        .........
clean:
        rm -f $(TARGET) $(OBJ) core *~
```

さらに、make では暗黙の生成ルールというものがある。例えば C ソースファイルからオブジェクトファイルを生成するためのコマンド系列を make は情報として持っており、実行コマンド行に何も指定してない場合は、暗黙の生成ルールが適用される。例えば、

```
main.o: inc.h
```

だけが書かれているような場合は、main.o の生成には暗黙のルール[15]が適用される。C ソースファイルからオブジェクトファイルを生成するための暗黙のルールを変更したい場合は次のようなエントリを Makefile に加えることで可能である。

```
.c.o:
        cc -c $<
```

このエントリは".c"が付いたファイルから".o"の付いたファイルを生成するためのルールを表す。

以上説明してきたマクロや変数などの機能を使うと、前述の Makefile は次のように書くこともできる。

```
TARGET = cmd
OBJ = main.o sub1.o sub2.o
$(TARGET): $(OBJ)
        cc $(OBJ) -o $(TARGET)
$(OBJ): inc.h

print: inc.h main.c sub1.c sub2.c
        pr $? | lpr
        touch print
clean:
        rm -f $(TARGET) $(OBJ) core *~
```

この Makefile を記述する作業は以外と面倒なものであるが、開発されるソフトウェアが大きくなればなるほど、そ

[14] UNIX のすべてのファイルにはもっとも最近アクセスされた時刻、もっとも最近変更された時刻などが記録されている。厳密にいうと、make は最終目的物を作るために必要なファイルの関係をすべて調べ、make は目的物と必要物の変更時刻を比較し、目的物よりも必要物の変更された時刻が後の場合は、そのコマンドが実行されることになる。

[15] この場合実行されるコマンドは cc -c main.c となる。

れによって得られる利益は非常に大きい。特に生成の手順が複雑になればなるほど、make はその実力を発揮する。

l i n t

C ソースプログラムの文法チェックだけを行うツールとして lint がある。lint はアセンブリファイルやオブジェクトファイルの生成を伴わないので処理が早く、また、チェックがコンパイラに比べ非常に厳しく行われるため、サイズの大きなファイルの文法チェックや発見しにくいバグのチェックに使うことができる。また、移植上問題になりそうな部分についての検査も行うので、ソースファイルを作成した時に lint で文法検査を行っておくのが望ましい。

lint の動作を C ソースファイルの中からも制御することができる。これは次のようなコメント文を挿入することで行える。

`/* NOTREACHED */`	制御が到達しない文に関する検査を行わない。
`/* VARARGSn */`	以降の関数宣言における引数の個数に関する検査を行わない。n を指定した場合は、最初の n 個の引数は検査される。
`/* ARGUSED */`	この宣言がなされた次の関数に対しては、その関数内で使用されない引数の検査を行わない。
`/* LIBTLIBRARY */`	ファイルの先頭に記述すると、そのファイルで使用されていない関数に対するメッセージを使用しない。

エラーメッセージの処理

UNIX には error というツールが用意されている。これは、コンパイル時のエラーメッセージを、ソースファイル上のエラーが発生した場所にコメントとして挿入するコマンドである。例えば、a.c というファイルのコンパイル時のエラーメッセージや make の実行時のエラーメッセージを挿入したい場合は次にようにすればよい。

```
% cc test.c |& error
```

```
% make |& error
```

-v オプションを付けると error の実行後、エラーがあったファイルに対してのみ vi が起動されるので、大量のファイルをコンパイル・デバッグする場合非常に便利である。

error はある程度便利なツールといえるが、その最大の欠点はソースコードを直接変更してしまう点にある。扱われるエラーメッセージが単純な時はよいが、C コンパイラがしばしばする一箇所の間違いによって引き起こされる連鎖反応的なエラーメッセージの出力に関しては、そのまま挿入してしまうので、関係の無いエラーメッセージまでが挿入されることになる。

デバッガ

ソフトウェア開発においてもっとも時間と手間がかかるのがデバッグである。デバッグの作業をサポートするツールとしてデバッガがあり、UNIX ではデバッガとして adb, dbx が用意されている。adb は機械語レベルでの汎用デバッガであり、バイナリファイルに直接パッチを当てたり、あるいはカーネルのデバッグなどに使われる。一方、dbx はシンボリックデバッガで一般利用者はこのデバッガを使うことが多い。4.1BSD までは sdb というデバッガが UNIX には用意されていた[16]が、4.2BSD 以降は sdb を拡張した dbx が標準で用意されている。

dbx を利用してデバッグするにはソースファイルをコンパイルする時に -g オプションを付けてコンパイルしておかなければならない。このオプションを指定してできたオブジェクトファイルから生成された実行形式ファイルを用いて dbx を起動する。

デバッガは実行状態でのプロセスのメモリ状態をそのままファイルに書き込んだ core ファイルを利用している。作成したプログラムを実行してみておかしな状態になった時に、しばしば UNIX は core ファイルを作って実行を停止する。実行形式ファイルを作成する時に -g オプションを付けておいた場合には、どの段階で core を作ったかをデバッガを使って調べることができるので、デバッグ中はむやみに core ファイルを消すのは得策ではない。

[16] System V 系の UNIX ではシンボリックデバッガは未だに sdb だけしか用意されていないことが多い。

`run`	プログラムの実行を開始する
`cont`	実行の再開
`step`	1 行づつ実行する。関数呼び出しがあった場合は実行を停止する。
`next`	1 行づつ実行するが、関数呼び出しがあっても停止しない。
`call` *proc*	call a procedure in program
`quit`	dbx の終了
`print` *exp*	式 *exp* の値を出力する。変数の値を見るのに使うことが多い。
`set`*var*`=`*value*	変数 *var* に値 *value* をセットする。これはプログラム実行中の場合にのみ利用可能。
`stop at` *line*	ソースファイルの *line* 行目にブレークポイントを設定する。*line* − 1 行目の実行が終了した段階で停止する。
`stop in` *proc*	関数 *proc* が呼ばれた段階で停止するようブレークポイントを設定する。
`trace` *line#*	ソースファイルの *line* 行目にトレースを設定する。
`trace` *proc*	関数 *proc* の呼び出しに対してトレースを表示する。関数 *proc* が呼び出されると、その関数名と引数の値が表示される。
`trace` *var*	変数 *var* の変更に対してトレースを設定する。変数 *var* の値が変更されると、新しい値が表示される。
`trace` *exp* `at` *line#*	プログラムの実行が *line* 行目に達した時の式 *exp* の値を表示する。
`file`	現在対象としているファイルを変更する。引数が無い場合は現在対象としているファイルの名前を出力する。
`list` *line*, *line*	ソースファイルの指定された行を表示する。
`where`	現在対象としている関数(active procedure)がなんであるかを表示する。
`status`	現在設定されているトレースとブレークポイントを表示する。この時にブレークポイントの番号などが出力される。
`delete` *number*	現在設定されているトレースやブレークポイントを解除する。引数にとしてブレークポイントなどの番号を指定する。
`whatis` *name*	*name* で宣言されている変数や関数の型を表示する。
`whereis` *name*	*name* で宣言されている変数や関数がどの関数中で使用されているかを表示する。
`sh` *cmd*	コマンド *cmd* をシェルで実行する。
`edit` *file*	ファイル *file* を編集する。
`alias` *name cmd*	*cmd* の alias を *name* とする。引数を与えない場合は現在設定されている alias のリストを表示する。

表 5: ｄｂｘのコマンド

dbx のコマンドを表5にまとめておく。

dbx は起動時に".dbxinit"というファイルを読み込んで、その中の設定を初期設定とする機能を持っている。この中で、ブレークポイントの設定や alias の設定をしておくと便利である。このファイルはまずカレントディレクトリを探し、無い場合はホームディレクトリを探すようになっているので、ソフトウェア毎に設定をすることができる。

dbx はプロセスが fork()などを使って新たなプロセスを生成した場合には、新しいプロセス側のデバッグができないとか、signal の操作がうまくいかないといった欠点を持っているが、一般的なソフトウェアのデバッグでは非常に有効なツールである。

C 以外の言語

UNIX では C 言語以外にも数多くの言語が用意されている。

Yacc/Lex

yacc/lex は C プログラムジェネレータであり、構文解析や字句解析を行うプログラムを生成する。yacc[14]はコンパイラ・コンパイラで、BNF から構文解析を行う C 言語の関数[17]を生成する。一方 lex[15]は字句解析を行う関数を生成するものである。この lex/yacc を合わせて使うことによって、簡単な言語系のコンパイラを自作することが容易になった。

FORTRAN77

FORTRAN77[16,17]も標準で用意されている。UNIX での FORTRAN の最大の特徴は C のプログラムとのリンクが可能であることであろう。すなわち、入出力部分は C 言語で作成し、計算部分は FORTRAN77 で作成するといったことが可能である。また、FORTRAN からも UNIX のシステムコールが利用できるようになっている。

[17] yyparse()

Ratfor

Ratfor[18]は FORTRAN77 のプリプロセッサである。Ratfor では C 言語なみの制御構造を使うことができ、FORTRAN77 と比べて記述しやすい言語といえる。

Pascal

Berkeley UNIX には Pascal[19]が標準で用意されており、これも C のプログラムとリンク可能である。Pascal では次のようなツールも用意されており、効率の良いプログラム開発環境が提供されている。さらに、文書清書システム TEX 等の一部のツールは Pascal を使って記述されており、これらのインストールや改良を行う場合は Pascal のプログラム開発環境を用意しておくことが重要である。

pc	コンパイラ。
pi	インタプリタ用仮想マシンコードを出力するトランスレータ。
px	インタープリタ。仮想マシンコードを実行する。
pix	インタープリタ。ただし、仮想マシンコードのトランスレーションも同時に実行する。
pxp	プリティ・プリンタ。
pdx	シンボリック・デバッガ。
pxref	クロス・リファレンスリストを作成するコマンド。

Lisp

Berkeley 系 UNIX では Franz Lisp[20]が用意されている。Franz Lisp は Maclisp 系の Lisp である。インタープリタとして lisp 、コンパイラ liszt 、クロス・リファレンス・コマンド lxref が用意されている。Franz Lisp でも C や Pascal で書かれた関数を取り込むことが可能である。

その他の言語

これ以外に利用可能な言語として関数型言語 fp[21], Prolog (C-Prolog)、オブジェクト指向の考え方を反映した C++, Objective-C 等が利用可能である。

ソースファイル管理

作成したソフトウェアのリリース後は通常ソースファイルを管理することになる。これをバージョン管理(あるいはリビジョン管理)というが、バージョン管理では最新のバージョンのソースファイルだけでなく、すべてのリリースのソースファイルと、リリース間の改良・変更の状態の保存が必要となる。

また、大規模ソフトウェアを開発する場合や、複数のプログラマーで一つのソフトウェアを開発する場合は、開発過程においてソースファイルの管理の必要性が生じる。例えば、プログラマAはライブラリを作成し、プログラマBがそのライブラリを使用してソフトウェアの開発をする場合を考える。開発過程においては、Aは開発したライブラリをBに渡し(すなわちグループ内でリリースする)、Aはその後デバッグや改良を続けることになる。一方、Bはそのライブラリを使用してソフトウェアの開発をすることになるが、開発をしている段階でライブラリのバグを発見した場合、BはAにそのバグがどのようなものかを伝え、改修を依頼することになる。この時、AがBに渡したライブラリのソースをAもBも保存していない場合、Bが発見したバグをAの側で見つけることができないとか、Bの発見したバグを直すことが難しくなる場合が多い。このように、開発グループ内でソフトウェアのリリースが発生する場合も、そのグループ内部でのバージョン管理が重要となってくる。

もっとも簡単なバージョン管理は、作成したファイルをバージョン毎にすべてそのままの状態で保存することである。この方法は可能ではあるが、非常に多くのディスク空間を消費するので好ましい方法とはいえない。エディタの章で説明した差分保存をするとディスク空間は少なく抑えることができるが、あるバージョンの状態を復元するのに繁雑な処理をすることになる。UNIX では差分保存の方法を用いて、特定のバージョンの取り出しやバージョンアップなどの処理を自動化したRCS (Revision Control System)というシステムを用意している。[18]。

RCS では差分保存のデータをまとめているファイル(RCS

[18]System V 系では SCCS (Source Code Control System) が一般的であり一部の Berkeley 系 UNIX(例えば SUN-OS)でも SCCS の利用が可能である.

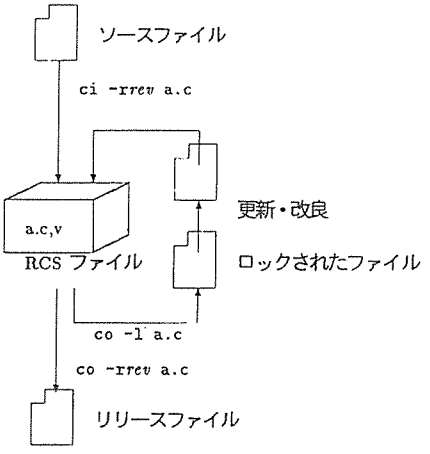

図 4: ＲＣＳファイルの操作

ファイルと呼ぶ)は",v"が付けられている。例えば"test.c"のRCS ファイルは"test.c,v"となる。RCS ファイルは元のソースファイルと同じディレクトリに作られるが、もしもカレントディレクトリにRCS という名前が付けられたディレクトリがあると、その中にRCS ファイルを作成する。これによって誤操作によってRCS ファイルを消去してしまう可能性をかなり減らすことができる。

RCS ファイルに対する基本操作は ci (check in) と co (check out) という 2 つの操作である。ci は新しいリビジョンをRCS ファイルに登録し、co はRCS ファイルから特定のリビジョンを取り出す(図4)。ソースファイルを作成したならばciを使ってRCS ファイルを作成する。この時リビジョン番号は-rrevオプションを使って指定することができる。何も指定しない場合は1.1になる。coでは、リリースするファイルを取り出す。この時もリビジョン番号を-rオプションで指定することができる。何も指定しない場合は、最新のリビジョンが取り出される。ソースファイルの更新をする場合は、まずRCS ファイルをcoで-lオプションを使ってロックしたリビジョンのファイルを取り出す。現在の最新リビジョンが1.5の場合はリビジョン1.6がロックされることになる。これによりRCS ファイ

ルには次のリビジョン(この場合だと1.6)が現在作成中であるという情報が記録される。取り出したファイルを変更し、それを新たにRCSファイルに登録する場合は、ciを使う。これにより新しいリビジョンが登録される。

RCSファイルに記録されている各リビジョンの記録を見るにはrlogというコマンドを使う。これを使うと、各リビジョンが登録された時刻などの情報が表示される。

ソースファイル中にリビジョン情報などをコメント等として入れたい場合はRCSで決められているキーワードを入れておくことで可能である。RCSでは表6ようなキーワードが用意されている。これらはリビジョン登録時に、対応する情報を表す文字列に置き換えられる。

例えば、Cプログラムソースファイルの先頭に、

```
static char rcsid[] = "$Header$";
```

としてあると、ciをした後にはソースファイルにはRCS標準ヘッダに置き換えられる。これをコンパイルした場合、オブジェクトファイルや実行形式ファイルにこの情報が書き込まれる。ファイル中のRCSキーワードを捜し出して、表示するidentというコマンドが用意されており、このようにしておくとソースファイル、オブジェクトファイル、実行形式ファイルにリビジョンが記録されることになる。オブジェクトファイルのバージョン検査に利用でき、非常に便利である。

関連文献

UNIXの解説文献は非常にたくさん出版されており、それらの中でUNIXのソフトウェア開発環境について解説しているものも数多くある。ソフトウェア開発環境を整備していく上で、各種の文献から情報やノウハウを得ることも重要である。

文献[22]は、UNIXを実際に作ったKernighanによって書かれたもので、UNIXにおけるシェルプログラムからCプログラムの開発までを広範囲に解説したもので、UNIXを使うものにとってはバイブルと言えるようなものである(これを日本語に訳したものが文献[23]である)。文献[24]はソフトウェア開発に使用できるUNIXのツール群(SCCS, Cコンパイラ, make, awkなど)についてきめ細かな解説をしている。また、文献[25]はBerkeley系UNIXのツールの使い方、環境設定について、ネットワーク機能の利用方法等の解説をしており、UNIXで本格的にソフトウェア開発を行うパワーユーザや計算機管理者向きの文献といえよう。文献[26]はC言語とUNIXの環境について日本語で解説してあり、入門書としては非常に良い。

UNIXにおけるC言語については、やはりKernighanが書いた文献[27,28,29]がバイブルであろう。文献[28]はANSI-Cをベースとした C言語について解説がなされており、文献[27]と内容を比較してみるのも非常に有意義である。文献[30,31]はUNIXのシステムコールを利用してツールを開発する際には非常に参考になる。この文献ではUNIXのシステムコールについて詳細な解説がなされている。文献[32]では計算機に依存しないCプログラムを書くためのノウハウが紹介されており、また、UNIXのバージョン間でのライブラリの違いなどについて詳細な解説がなされている。

UNIXのカーネル内部の構造を知るには[19]文献[33]が最適である。文献[34]ではUNIXのカーネル内部の構造からネットワーク機能・ウィンドウシステムまで解説がなされており、UNIXワークステーションの内部構造を知るには適している。

このように多くの文献が出版されているが、やはりBerkeleyからディストリビューションされているマニュアル群を読破するのが、ツールの使い方等を知る場合には一番確実な方法である。また、解説文献にはないいろいろな機能を知ることもできる。UNIXでソフトウェア開発をする方はマニュアルを読むことをお勧めしたい[20]。

おわりに

本稿ではUNIXにおいてソフトウェア開発を行う際に使われるツール群の紹介と便利な使い方を解説した。

UNIXでのソフトウェア開発は、UNIXで用意されているツールを、どうやって効果的に利用するかという点が鍵に

[19]一番の方法はカーネルのソースを読むことである。UNIXのカーネルはかなり小さいのですべてを読むのも不可能ではない。

[20]4.3BSD UNIXのマニュアルについてはUSENIXの会員は購入することができる。

`$Author$`	リビジョンを check in した人のログイン名
`$Date$`	リビジョンが check in された時刻
`$Header$`	RCS の標準ヘッダ (RCS ファイル名、リビジョン番号、check in した日時、check in した人のログイン名、ステート)
`$Locker$`	現在の RCS ファイルがロックされている場合、ロックした人のログイン名
`$Logs$`	check in の時に入力したログメッセージ
`$Revision$`	リビジョン番号
`$Source$`	RCS ファイルのフルパス名
`$State$`	RCS ファイルのステート (ステートは `ci -s`*state* で与えられる。デフォルトは exp)

表 6: ＲＣＳのキーワード

なっている。事実、UNIX で多くのソフトウェアを開発しているプログラマは、大抵の場合 UNIX のツールについて非常に多くの知識を持っており、ツールを手足のように使っている[21]。本稿ではツール群のほんの一部を紹介したに過ぎず、このほかにも多くの有用なツールが用意されている。本稿が読者の開発環境の改善の助けになれば幸いである。

References

[1] 藤川 和利, 山口 英, 馬野 元秀. Unix の簡単な使い方. 大阪大学大型計算機センターニュース, Vol.18, No.3, 1988.

[2] Brian W. Kernighan. A tutorial introduction to the UNIX text editor. *UNIX User's Supplementary Documents*, USD:12, April, 1986.

[3] Willam Joy and Mark Horton. Ex reference manual version 3.7. *UNIX User's Supplementary Documents*, USD:16, April, 1986.

[4] William Joy and Mark Horton. An introduction to display editing with vi. *UNIX User's Supplementary Documents*, USD:15, April, 1986.

[5] Richard Stallman. *GNU Emacs Manual.* Free Software Foundation, Cambridge, MA, March, 1987.

[6] Richard Stallman (竹内郁雄・天海良治監訳). *bit* 別冊:*GNU Emacs* マニュアル. 共立出版, 1988.

[7] Jonathan Payne. JOVE manual for UNIX users. *UNIX User's Supplementary Documents*, USD:17, April, 1986.

[8] S. I. Feldman. Make – a program for maintaining computer programs. *UNIX Programmer's User's Supplementary Documents*, PS1:12, April, 1986.

[9] S. C. Johnson. Lint, a C program checker. *UNIX Programmer's Supplementary Documents*, PS1:9, April, 1986.

[10] J. F. Maranzano and S. R. Bourne. A tutorial introduction to ADB. *UNIX Programmer's Supplementary Documents*, PS1:10, April, 1986.

[11] Bill Tuthill and Kevin J. Dunlap. Debugging with dbx. *UNIX Programmer's User's Supplementary Documents*, PS1:11, April, 1986.

[12] Walter F. Tichy. An introduction to the revision control system. *UNIX Programmer's User's Supplementary Documents*, PS1:13, April, 1986.

[13] Eric Allman. An introduction to the source code control system. *UNIX Programmer's User's Sup-*

[21] この際たるプログラマがハッカーである。

plementary Documents, PS1:14, April, 1986.

[14] Stephen C. Johnson. Yacc: Yet Another Compiler-Compiler. *UNIX Programmer's Supplementary Documents*, PS1:15, April, 1986.

[15] M. E. Lesk and E. Cshmidt. Lex – a lexical analyzer generator. *UNIX Programmer's Supplementary Documents*, PS1:16, April, 1986.

[16] S. I. Feldman, P. J. Weinberger, and J. Berkman. A portable Fortran 77 compiler. *UNIX Programmer's Supplementary Documents*, PS1:2, April, 1986.

[17] David L. Wasley and J. Berkman. Introduction to the f77 I/O library. *UNIX Programmer's Supplementary Documents*, PS1:3, April, 1986.

[18] Brian W. Kernighan. RATFOR – a preprocessor for a rational fortran. *UNIX Programmer's Supplementary Documents*, PS2:8, April, 1986.

[19] William N. Joy et.al. Berkeley pascal user's manual. *UNIX Programmer's Supplementary Documents*, PS1:4, April, 1986.

[20] John K. Foderaro, Keith L. Sklower, and Kevin Layer. The FRANZ LISP manual. *UNIX Programmer's Supplementary Documents*, PS2:9, April, 1986.

[21] Scott Baden. Berkeley FP user's manual. *UNIX Programmer's Supplementary Documents*, PS2:7, April, 1986.

[22] Brian W. Kernighan and Rob Pike. *UNIX Programming Environment*. Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, 1984.

[23] Brian W. Kernighan, Rob Pike (石田晴久監訳). *UNIX* プログラミング環境. アスキー出版局, 1986.

[24] 坂本 文. *UNIX* ツール ガイドブック. 共立出版, 1986.

[25] 村井 純, 井上 尚司, 砂原 秀樹. プロフェッショナル *UNIX*. アスキー出版局, 1986.

[26] 松山 泰男. *C*言語と *UNIX*. 日刊工業新聞社, 1986.

[27] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, 1978.

[28] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language - Second Edition*. Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, 1988.

[29] Brian W. Kernighan, Dennis M. Ritchie (石田晴久訳). プログラミング言語 *C-UNIX* 流プログラミング書法と作法. 共立出版, 1986.

[30] Marc J. Rochkind. *Advanced UNIX Programming*. Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, 1985.

[31] Mac J. Rochkind (福崎俊博訳). *UNIX* システムコール・プログラミング. アスキー出版局, 1987.

[32] J.E.Lapin. *Portable C and UNIX System Programming*. Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, 1987.

[33] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, 1986.

[34] 村井 純, 砂原 秀樹, 横手 靖彦. *UNIX* ワークステーション *I*. アスキー出版局, 1987.