make による構築と検査

make の概略

make の誕生

make は 1976 年 4 月、ベル研究所のスチュアート・フェルドマン(Stuart Feldman)氏が考案した。既にベル研究所では c 言語が使われていた。ベル研究所では大きいシステムを小さい要素に分割する流儀があり、UNIX もこれに従う。ソースファイルが小さいとそれぞれは管理しやすいが、数が多くなると別の問題が発生する。手動で構築すると見落としが発生し、自動的なバッチ処理とすると時間がかかった。ソースファイルとオブジェクトファイルのタイムスタンプを比較し必要となる物だけ更新する機構として make が誕生した。

余談ながら 1976 年 4 月 1 日にはアップル社が創業した。同年夏に NEC からワンボード・マイコン TK80 が発売された。

その後 make は UNIX とともに普及し、makefile はソースコードから実行形式を構築する手順を記録する標準書式となった。

以下のようなプロジェクトの保守管理の手順も併せて makefile に記録する慣習が定着した。

- 中間生成物と実行形式を消去してソースツリーを清新な状態にする
- 変数、関数の相互参照関係（クロスリファレンス）等の解析
- ソースコードの整形
- ソースファイルの印字
- 種々のテスト
- ドキュメントの整形、書式変換
- tar や zip 形式のアーカイブ作成

簡単な makefile

make では makefile というテキストファイルに入力出力のファイル名と出力を得るための命令を記述する。

```
hello: hello.c
	gcc hello.c -o hello
```

上の行の：の左側が出力でこれは「ターゲット」と呼ばれる。：の右側は入力で出力が依存するファイルを列記する。下の行はターゲット生成のための命令である。左側の空白はタブである。依存関係を記述する行と処理を定めた行を合わせて「ルール」と呼ぶ。

この例では hello（ターゲット）が存在しなければ生成する。あるいは hello が hello.c と比較してより古ければ生成し直し、新しければ何もしない。hello.c が存在しない場合はエラーとなる。

実行形式を生成するにはコマンドラインで以下のように指令する：

```
make hello
```

一般的な makefile

次は一般的な例で構築を複数の段階に分けて行う。c ソースファイルを makefile の中でソースファイルを cat で併合するというのはあまり見ないでしょうが、後で実用例を紹介します。

```
hello: hello.o
	gcc hello.o -o hello

hello.o: hello.c
	gcc -c hello.c -o hello.o

hello.c: head.c body.c
	cat head.c body.c > hello.c

head.c:
	echo '#include <stdio.h>' > head.c

body.c:
	echo 'int main(){ puts("Guten Tag!"); }' > body.c

message:
	./hello

clean:
	rm -f head.c body.c hello.o hello
```

最後2つのルール、message と clean はその名のファイルを生成しない。これは「擬似ターゲット」と呼ばれる。

上記のように makefile にテキストを埋め込み echo 等でターゲットを生成する方法には落とし穴がある。一度構築して、文字列 Guten Tag を変えて hello.c を生成するとわかります。

make は機能豊富だが学習しにくい

make には高度な機能が多数ある。

- 変数 マクロ
- 選択的実行 (if 文)
- 関数
- 他の Makefile の実行（階層ディレクトリ構造に対応）

こうした機能を持つ make は言語である。しかし機能豊富になったことで make は複雑に、学習しにくいものにしてしまった。UNIX を使いソフトウエアを開発するのにはシェルやエディタ、デバッグツールの操作を学ぶ必要がある。そこに make が加わると負担が大きい。

ここでは基本的な機能を使った例により解説する。

make が複雑難解になった背景については付録1を参照。

make を利用した検査

スタブの利用

サブルーチン（関数）を単体で検査する場合、スタブ(stub)と呼ぶ小規模な仮設のメインルーチンをリンクする方法がある。出荷する製品に外部器機との通信、手動マウス操作など検査の自動化の妨げとなる処理がある場合、通常のメインルーチンによらずサブルーチンを呼び出すスタブを用意する。自動化による生産性向上は顕著なのでこれは是非覚えておきたい技術である。以下の例か、ご自作の簡易な makefile を活用して実感して頂きたい。

```
ktest: stub.o kaijo.o
	gcc stub.o kaijo.o -o ktest

stub.o: stub.c
	gcc -c stub.c -o stub.o

kaijo.o: kaijo.c
	gcc -c kaijo.c -o kaijo.o

check: check1 check6 check1-12 check-invalid

# 出力を文字列として比較
check1: ktest
	test `./ktest 1` = 1

# 出力を整数値としてシェル算術の結果と比較
check6: ktest
	test `./ktest 6` -eq $$((1 * 2 * 3 * 4 * 5 * 6))

# 出力をファイルに格納して、予想される正しい内容のファイルと比較
check1-12: ktest
	for n in `seq 12`; do ./ktest $$n; done > output
	cmp -s output output.ok && rm output

# 終了コードを検査
check-invalid: ktest
	./ktest 1 2 3 || test $$? -eq 1

demo1-20: ktest
	for n in `seq 20`; do ./ktest $$n; done

clean:
	rm -f ktest stub.o kaijo.o output
```

文字列、整数値を比較する時は test、2つのファイルが一致するか調べるには cmp を使う。いずれもコマンドライン命令である。test はシェルの組み込み命令として実装されていることが多い。

上記例 check6 と check-invalid では$$が見られる。make の変数との区別のためシェルの$は二重にする。構築した実行形式（上の ktest)はパスを明示しないと動作しない。細かい注意点が多い。慣れるまで試行錯誤を続けて下さい。

test や cmp を make の中で行わなず、ファイルに結果出力するなどして make の外で（あるいは別の make で）判定する方法もある。この方式はコマンドの実行に問題があったら make がエラーで終了し、出力内容に問題があったら外部判定で確認され区別されるという利点がある。上記の例なら make demo1-20 を利用すると良い。

上記で利用可能な stub.c は付録3として添付。kaijo.c は自作して下さい。

ソースファイルを部分の併合で作成する例

ソースファイルを分割するのが望ましくないような場合は次のような方法を用いる。

出荷版とテスト版を cat で作り分ける:

```
# 出荷版
product: main subroutines
	cat main subroutines > product

# テスト版
tester: stub subroutines
	cat stub subroutines > tester

test1:
	./tester < input1 > output1
	cmp -s output1 output1.ok
```

この他に出荷版ファイル（上の例なら product）の中に標識となる文字列を注釈という形で仕込んでおいて csplit や sed などでファイルの必要部分を切り出す方法もある。cat の利用が足し算ならこれらの利用は引き算となる。

エディタで上記例のテスト版を作成することも可能だが手作業は間違いのリスクがある。また手動操作が無ければ上記例のように makefile の中に記述し作成を自動化できる。

bzip2 パッケージの Makefile

圧縮ソフト bzip2 のパッケージは configure が無く Makefile の見通しが良い。

基幹パッケージでは Makefile は自動生成され、解析しにくいものがほとんどであり、これは例外である。

https://www.sourceware.org/bzip2/

https://www.sourceware.org/bzip2/downloads.html

このパッケージを入手して解析し、独自作成したテスト項目を加えるなど少し改変して試行すると良いだろう。

GNU awk (gawk)のテスト群

GNU awk (gawk)では tests というディレクトリにテストのスクリプトと各々の実行で予想される出力が集められている。

Makefile を解析できなくてもこのパッケージのように tests（または同様の名前の）ディレクトリ内のファイルが参考になる場合がある。

http://savannah.gnu.org/projects/gawk/

https://ftp.gnu.org/gnu/gawk/

対話的処理のテストを支援する道具

対話的処理が必要な場合以下のパッケージを使用すると良いとされる。gcc などの検査に使われていると言うが、中身について詳しく知らないので名前だけ挙げておく。

- DejaGNU
- Tcl/Tk
- Expect

Netpbm の構築、検査とインストール

この部分では Netpbm の構築検査環境を概観した上で一般的な技術理解に資する各点を簡潔にお伝えする。

Netpbm の構築環境の課題

Netpbm は画像書式変換、処理のプログラム集で歴史は古く、1980年代中頃に登場した。Netpbm は Linux(カーネル）より古い。環境の変化に合わせて構築も進化して来た。

構築には make を用いるが以下の課題がある:

- 多様な OS、環境で使われている。（GNU/Linux,BSD,MacOS,Windows）
- プログラムが 300 以上と多数ある。
- 全てではなく一部のプログラムしか生成・利用しない利用者が多い。ディストリビューションによっては一部プログラムを割愛している。外部ライブラリが入手困難になり環境次第で生成不可能なプログラムもある。テストとインストールはこれに対応しなければならない。
- 画像ファイルは大きくなるもので様々な入力パターン、全てのテストの出力結果を用意したらパッケージが肥大化する。
- 画像データは大きさがファイルごとに異なる。c の malloc 関数を多用していて検査が必要。

プログラムが多数に及ぶのはそれぞれの機能が限定されていることの反映でもある。それぞれのプログラムは検査しやすいという利点をもたらしている。

Netpbm におけるインストールとテストの進行

プログラム構築後、make package により仮設ディレクトリにまとめる。これは主要パッケージでは例を見ない方法である。プログラムが多数で /usr/local/netpbm/ のように Netpbm 専用のディレクトリを作成してそこにまとめて格納していた利用者が過去には多くいたことを反映している。最近では実行形式を一般的な/usr/local/bin/ や /usr/bin/ にインストールする利用者が多い。

テストは3通りの対象を指定可能

1. 仮設ディレクトリに集約された実行形式
2. ソースツリー上の実行形式
3. インストールされた実行形式（システム通常のパス）

デフォルトは1.だがほとんどのパッケージで行われている一般的な方法は2である。開発中は2.が便利だが実行形式がソースツリー上に散在していて複数のプログラムを組み合せるテストが大多数なので工夫を要した。

一部のプログラムだけの構築、テストに対応する makefile はどうしても複雑になる。

システムで常用されるプログラムのテスト特有の問題

システムで常用されるプログラムのテストでは既にインストールされ、通常使用されている版と新たに構築したテスト対象版との区別が必要となる。これが最も顕著になるのは ls mv rm mkdir など基本コマンドを提供する coreutils パッケージであろう。PATH に工夫したりテストするプログラムの名前を（前後に文字列を追加するなどして）変えるかする。これはテストが複雑で判読しにくくなる原因である。判読しにくいと参考に向かない。

Netpbm におけるテストの方法

個別の検査は大別して以下のようなパターンに分けられる。

これは他の開発プロジェクトでも参考になると思う。

1. コマンドを実行して予想される結果と照合　（「回帰テスト」と呼ぶ）

   ハッシュ関数で結果を縮約することがほとんど
2. コマンドを実行して出力を解析
3. 往復テスト 変換と逆変換

- 可逆変換　もとの入力と往復後出力が一致するか照合
- 不可逆変換 往復後出力がどの程度変化したかを測定

4. 異なるコマンドで同一の出力が得られるか調査
5. 不正なコマンドにてエラー終了、出力無しを確認
6. valgrind 監視下でメモリーチェック
7. ファジング（Fuzzing）

最初は1.と3.が多かった。その後5.が増えた。

回帰テスト

回帰テスト(regression test)では改変によって動作が変わっていないかを調べる。方法はいくつかある。一例として、旧版と新版を同じ条件で動かして出力結果を比較する。

回帰テストは改版によって動作に影響が出ていないかを確認するだけだが、この確認は大事である。またどの部分が無関係か早急に判明するので異常の原因の絞り込みに有効である。不具合は得てして思わぬ所に潜んでいる。当たり前の事象をたくさんチェックするのが良い。

例として coreutils の seq コマンドのテストをするなら以下のように makefile に記述する。expected-output はあらかじめ作成しておく。

```
check:
	seq 10 20 250 | cmp -s – expected-output

# 比較するファイルが大きくなる場合はハッシュ関数で縮約する方法もある。
# 文字列に空白があるので引用符で囲む。
check-md5sum:
	test "$$(seq 1000 | md5sum)" = \
	"53d025127ae99ab79e8502aae2d9bea6  -"
```

往復テスト

往復テストではコマンドとその反対の働きをするコマンドを組み合わせる。

以下に gzip 圧縮ソフトの往復テストの一般的な例を示す。

```
check-gzip:
	gzip -c test-input | gunzip -c | cmp -s test-input -


# 異常終了した場合は出力を保存
check-gzip2:
	gzip -c test-input | gunzip -c > test-output
	cmp -s test-input test-output && rm test-output
```

これを圧縮ファイルを始点終点とするとうまく行かない。gzip は圧縮率を選択指定できて出力が一意的に定まらない。出力の一意性はテストの設計において意識する必要がある。

```
# 失敗することが多い
check-gunzip:
	gunzip -c test-input.gz | gzip -c | cmp -s test-input.gz -
```

テストがきっかけで追加されたプログラムや機能

テストを作成していると既存のプログラムの不便な側面が見えて来るもので、それを踏まえて機能が追加され、また新たなプログラムが作成された。

一般的に参考になる事例を挙げる:

乱数

結果出力の一意性を保つために乱数の種を設定するオプションを追加。さらに OS の違いに影響されないよう、システム乱数を使うのをやめ、Mersenne Twister のコードを導入してパッケージ自体に乱数発生ライブラリ関数を持たせた。

限定的出力

Netpbm には画像を拡大縮小したり、上下左右に縁を追加する、画像の一部を切り出すなどして入力と出力で画像の大きさが変わるプログラムが多数ある。(pamscale, pamenlarge, pnmpad, pamcut 等)これらについて画像自体を出力する代わりに大きさだけを報告するオプションを追加した。その部分のテストが迅速になるとともに、新たなテストの作成が可能になった。

grep には一致した行の数だけ報告する -c というオプションがあるがそれに似ている。

特異な値の扱い

pnmpsnr は2つの入力画像の差異を測定して S/N 比率(sound noise ratio)で報告する。単位はデシベル(dB)でこの値が大きいほどよく一致している。完全に一致すると無限大となってしまうのでその時はデシベルを出力せず、一致を通知する仕様だった。

しかし一致した場合を特異な例とすると呼び出すプログラムはそれに対応する必要から複雑になる。報告するデシベル値の上限を利用者の指定できるようにし、それを超過したらその値を出力するオプションを追加した。

パイプラインを堰き止める

パイプライン処理は効率的であるが、異常な入力への対応が課題である。入力に瑕疵があり、処理するプログラムがつかえると中途半端な出力が生じることがある。そこで入力画像に問題が無いかを検証し、合格した場合だけ出力する(パイプラインの後段にデータを流す) pamvalidate というプログラムを新たに作成した。

パイプライン処理を多用している方には参考になると思う。

テスト用画像の作成

乱数を使って既存の画像を撹乱する pamshuffle、長方形の画像の画素を横1列に並べて別の寸法の長方形に畳み直す pamrestack 等が追加された。小さいプログラムを複数用意するとその組み合わせで多様な出力が可能になる。

同じく追加された pbmnoise は白黒2値の任意の大きさのごま塩をふったような画像を乱数を使い効率的に生成する。白黒の比率は分母を2のべき乗として任意に指定できる。デフォルトは白黒半々。黒の割合を 1/8,15/16 にする方法はすぐわかるが 3/8, 5/16 などの割合が欲しいとなると頭を使う。読者の皆さんにアルゴリズムをお考え頂きたい。

valgrind

valgrind は広く利用されている開発支援の道具集である。いくつかの道具があるがメモリー参照の問題を検出する memcheck が最も有用である。c 言語では malloc した領域からはみ出す読み書き、malloc していない領域の free、値を書き込み定めていない変数の値の参照を検出する。Segmentation Fault（参照範囲例外）で実行が停止した時の原因究明に役立つ。c、アセンブリの他、Rust などのプログラムも解析できる。

bzip2 というプログラムを検査するには次のように先頭に valgrind を追加して実行。

```
valgrind bzip2 -c input > compressed 2> valgrind.out
```

この例では通常の出力は compressed 、valgrind 出力は valgrind.out

検査対象プログラムが正常終了しても valgrind で問題が検出される場合、プログラムの終了コードが報告される。valgrind が異常を検出したかを確かめるには valgrind の出力（上記例では valgrind.out）を調査するか、valgrind に –error-exitcode=N オプションを与えて実行し、終了コードを valgrind 優先にする。

一般に valgrind テストは多くの時間を要する。また最適化して生成した実行形式では偽陽性（第一種過誤）の報告が度々見られる。問題箇所の特定にはデバッグシンボルが必要となるからコンパイル時に -g を指定しておく。

スタブを使った単体テストでも valgrind は利用できる。深刻な問題は早めに発見するのに越したことはない。また結合後では問題の原因の候補は多くなる。単体テスト段階から積極的に valgrind を活用すると良い。

Netpbm のテストでは make 発令に添加する valgrind を網羅的に適用するオプションがある。

Linux カーネルは valgrind で検査できないため専用の道具が用意されている。

ファジング (Fuzzing)

ファジングとは瑕疵のある入力を意図的に与えてプログラムをを実行する検査方法である。

正常な入力ファイルを用意し、乱数を用いて無作意に選んだ1バイトだけ値を変更することが多い。1バイト欠落させる、余計なバイトを挿入することもある。プログラムが異常な入力でも正しく対処できるかを見る。検査は valgrind の監視下で行うことが多い。

Netpbm パッケージ付属の検査ではこれは行っていない。外部のボランティアが担当。

Netpbm のプログラムは UNIX の流儀に従い対話的な応答を行わない。1秒で1件の異常なファイルを処理できるとすると、1万件処理に3時間ほどかかる。夜間にバッチ処理すれば良い。これがもし異常の報告に度に「続けますか？ yes/no」という質問が表示されてオペレーターが対応しなければならなかったら作業負担は大きくなり、同じ時間で検査できる件数は少なくなってしまう。

テストしやすい構成というのは大きな利点となる。

構築の再現性

Netpbm の構築で生成される実行形式にはバージョン情報とともに構築日時の刻印（タイムスタンプ）がつく。同じソースでも構築時刻が異なると異なる実行形式が生成されて再現性が無い。また構築日時よりソースの最終更新日時のタイムスタンプの方が望ましい場合もある。一般的慣習に従い、SOURCE_DATE_EPOCH という環境変数が定義されている場合はこれを参照することにした。

付録 1　make が学習しにくい理由

make は学習しにくい。それは以下のような理由による：

1. UNIX は複数の道具を組み合わせて使うと力を発揮するが、それぞれの道具に特徴があり、複数の言語の習得が必要。make の文法にも通ずるとなると学習負担が大きい。色々学習しなくて済むのでスクリプト言語が好まれ普及した。
2. POSIX make, GNU make, BSD make など多数の実用版があり、makefile の文法に違いがある。
3. 色々ある中で GNU make の利用が多い。それにより教書も出版されていてネットの情報も多いが GNU make は独自の機能が多くて複雑である。教書は稀にしか利用しない機能まで解説していて見通しが悪い。
4. シェルにも変数、選択的実行、関数がある。make の同種機能との区別が面倒。
5. makefile は実行順序を追跡しにくく、変数値の設定などわかりにくい。
6. GNU/Linux の主要パッケージは makefile が無い。ほんとんどの場合、自動生成される。自動生成された Makefile は理解困難。makefile の良い手本がなかなか無い。

1990年代前半にパッケージ管理システム（初期には dpkg 現在は apt や yum）が普及するまで、ディストリビューションというのは無く、GNU ソフトウエアはソースコードで配布されていた。GNU ソフトウエアは可搬性が高い（ポータブル）と評判が高かった。様々な OS の間には差異がある上、そこに GNU ソフトウエアが様々な組み合わせでインストールされていて環境は変化に富んでいた。それを各パッケージの configure というスクリプトで判定して makefile に渡す変数を生成していた。やがて make の上位の道具集として GNU autotools が整備され、Makefile は自動生成が一般的になった。

make は perl, python, javascript, ruby などより古い言語である。GNU make が成長した時期はワークステーションの性能が向上していて、他のいくつかの GNU のプロジェクトとともに意欲的な機能拡充が行われた。単機能で洗練された要素を組み合わせて行く UNIX の思想に反して肥大化と複雑化が進み、make のマニュアル、多くの実用 makefile、ともに見通しが悪くなった。こうなったことへの批判は根強い。

リーナス・トルバルズ氏が開発した Linux は厳密にはカーネル（中核部分）だけである。トルバルズ氏の率いる集団はカーネルの開発に専念していて彼はその外部はいっさいアプリケーションみたいな言い方をしている。これは本来の UNIX の定義とは違う。ともかく彼はカーネルの外の諸規格を制定する活動に積極的に関与しない姿勢である。カーネル以外まで俗に Linux と呼ぶが、人が主に触れるカーネルの外側の部分の規格になると絶対的な権威は存在しない。POSIX 標準が一定の支持を得てはいるがこれに縛られない例も多い。基幹要素のパッケージにおいても複数の選択肢が存在していて各ディストリビューションはそれぞれ選定している。差異を吸収する機構が不可欠であり、autotools と make はその中心にある。

Linux カーネル, gcc, glibc, X Window System など多くの基幹要素の構築においては make は GNU make と指定されている。BSD では当然 BSD make が好まれているが GNU 由来の物を始め、多くのパッケージの構築に必要なので GNU make も利用されている。

make 自身の構築という「種無し西瓜の種の入手」みたいな課題に GNU make と BSD make は対応している。これらは普通 make を使い構築するが、それが存在しない環境のために構築用シェルスクリプトが提供されている。これで仮の make を生成し、それを使い通常の構築を行うことが推奨されている。こうした工夫のおかげで GNU や BSD の make はそれが無かった環境にも入り、それに依存する他の基幹ソフトもソースコードから構築してインストールする道が開かれた。

近年 GNU autotools,GNU make の代替としていくつかの構築管理の道具が登場している。cmake と meson, ninja が代表的である。それらを使って構築するパッケージは増えてはいるが、ポータビリティという点では歴史のある GNU 系にまだ及ばない。GNU の構築体系に依存するソフトウエア資産は多く、それらの書き換えの動きは無い。

会社のプロジェクトで make を使うにあたりマニュアルや教書を最初から最後まで通読する必要はない。不要と思われる機能に遭遇したら飛ばして良い。個人的なプロジェクトでも makefile は更新頻度が低くなりがちで時間が経つと昔やったことを忘れてしまうなので高度な機能はなるべく避け、注釈を積極的に入れるべきである。一方で makefile 作成で遭遇する問題は GNU make の進んだ機能を使うと解決することが多い。その方がすっきりするなら高度な機能を活用すべきである。

付録２　makefile が無いパッケージでの一般的な構築手順

基幹ソフトのソースパッケージは一般に様々な OS・環境に対応可能なように作られていてダウンロードするファイル群には makefile は無く、自動生成される。makefile を検証したい場合は、次の手順により生成する。

一般的な GNU/Linux のパッケージの構築

.tar .zip 形式アーカイブ展開後に構築する場合

（Makefile.am が存在する場合）

```
./configure
make
make check
make install  # 管理者権限が必要
```

Makefile だけが必要なら最初の./configure で十分。

以上の全てを行うと実行形式ファイルはインストールされ利用可能になる。

なお上の configure は多くの場合オプションを与えることができる。静的動的ライブラリの選択、インストール先の指定、実行形式の名前の変更などが可能である。一般に configure --help とするとオプション一覧、影響のある環境変数が表示される。

git からソースを入手して利用する場合

（Makefile.am が無い場合）

```
automake
./configure
make
make check
make install
```

configure スクリプトも多くの場合 autoconf というプログラムによる自動生成である。筆者は生成を必要とするパッケージを見たことはない。自動生成された configure スクリプトは理解しにくいものです。

複数の makefile からの選択

一部のパッケージでは Makefile.linux Makefile.bsd Makefile.dos など名前の異るファイルが複数用意されていてリンクかコピーにより Makefile と改称して利用するよう求められる。こうしたパッケージは少数である。

Linux は Makefile を直接起動

Linux カーネルのソースには Makefile がある。configure に相当する設定は make oldconfig make menuconfig など、make から起動するいくつかの方法が提供されている。他の主要パッケージと違い対話的な設定が主流である。自動設定も可能で例えば「動作中のカーネルの諸設定をそのまま継承して新しく追加された項目は全て yes とせよ」のように指定できる。

この Linux カーネルの Makefile は解析し易いものではない。

make menuconfig などの設定項目は膨大で、全て調査して選定したら大変な作業となるが、カーネルとデバイスドライバの理解に資する良い経験になる。

付録３　stub.c

/* 整数の unsigned long int 型が 64 ビットの場合 20! で、32ビットの場合は 12! まで対応できる。*/

```
#include <stdio.h>
#include <stdlib.h>

extern unsigned long int kaijo (int const n);

int
main(int argc, const char ** argv) {

    unsigned long int n; /* 64 or 32 bits */
    char * tail;

    if (argc - 1 != 1) {
        fprintf(stderr, "error: wrong number of args: %d\n", argc - 1);
        exit(1);
    }
    else {
        n = strtoul (argv[1], &tail, 10);

        if (n==0) {
            if (argv[1] == tail || tail[0] != '\0') {
                fprintf(stderr, "error: %s is not a number\n", argv[1]);
                exit(2);
            }
            else printf ("%lu\n", kaijo(n));
        }
        else if (n > 20) {  /* 64 bits:20  32 bits:12 */
            fprintf(stderr, "error: %lu is too large\n", n);
            exit(3);
        }
        else printf ("%lu\n", kaijo(n));
    }
}
```

/* 階乗を計算する関数 kaijo()は自作して下さい。*/

筆者自己紹介

筆者が make を初めて使ったのはそれ以前の 1990 年頃、NTT の武蔵野研究所で使う自走ロボットの制御端末の開発を任された時でした。make による構築の管理によりコンパイルに要する時間を削減でき、少ない負担で頻繁に構築できるようになり、作業効率は向上しました。c 言語の assert 関数の活用も問題の早期検出に寄与していました。

また苦労して当時の非力なパソコンで動く GUI を作成しましたが納入先でどう実際に使っているのかを見聞きしてロボットの動作命令を複数記述したファイルを使う機能を追加しました。GUI はデモには良かったものの実際のテストにはバッチ処理機能の方が便利です。

同じ頃会社の先輩達は事務用ファクシミリの開発を行っていました。ソースはアセンブリ言語でした。単体試験を行っていた様子はほとんどなく、全体をリンクして実際の機材を操作しながら問題を探っていました。最初の目的として正常な送受信のルートを動くようにしようとしていると聞き、assert 関数を活用していた経験からこれは間違ったやり方だと思いました。初期は異常だらけに決まっています。アセンブリのプログラムは間違いがあると止まってくれず暴走します。異常が検知されたら停止するよう仕向け、操作履歴と停止時の状態が記録される機構をまず整備すべきだったでしょう。

問題が発見されるとパッチという方法で対処していました。パッチとは問題箇所の前に CPU のジャンプ命令を挿入して修正コードに飛んで帰還するというものです。パッチは局所的な問題の対処には適当であっても、システムの広い所に影響が及ぶ問題には不適です。現場の者はコードが混乱していると認識していました。作業効率は悪く、リーダーは「リンクし直せば改善すると思う」と言っていましたが、発注側がそれをさせてくれなかったのです。発注企業の管理者は新たにリンクをすることで新たな問題が発生し、テストを全てやり直すことになり時間とお金を浪費すると懸念していたのです。

もしテストの自動化が実現していたら事情はかなり違っていたでしょう。しかしそれには基本設計の段階から配慮する必要があります。後からではできることは限られてしまいます。

アセンブリ言語でパッチを書く職人芸には敬服します。しかしいくら難しい技術を習得しても効率が悪い開発では仕事が来なくなり、技術者の離職と会社の廃業が起きます。業界の外の人はプログラマの雇用不安定を見て、プログラミングとは習得に時間と努力を必要としない易しい技術と思ってしまうものです。更にプログラマに仕事を発注する方としてはこれを念頭に、専門用語に惑わされないで非熟練労働にふさわしい低賃金に抑えるのがシステム開発費用の削減に有効と考えるものです。こうした考えは払拭したいものです。

Netpbm の開発とメンテナンスに筆者は 20 年ほど前から参加しています。アルゴリズムの改良、機能の追加、多言語対応、問題点の発見と修正、検査の自動化を行って来ました。

Netpbm のプログラムは1990年代には不具合、不効率、限界が目立ち、リスクを理解した上で使う代物でした。多くの方の努力により品質が向上するとともに基幹ソフトウエアとして定着しました。サーバーで広く利用されるようになって、一層信頼性が求められるようになりました。検査の定式化と充実はこの要請に応えるものでもありました。

valgrind のような強力な検査用ユーティリティのお世話にもなっていますが、検査は基本が大事で何十年も変わっていないなと思います。

拙稿が皆様の待遇改善のお役に立つものでしたら光栄です。

2025年3月　漆畑晶