解　説

UNIXの限界と今後の進路†

木　村　泉††

1. 本文の立場

UNIX はすばらしいオペレーティング・システムである．筆者はいまそれを，勤務先と自宅の両方で使っており，それが使えなかった2，3年前との大きな違いを痛感している．

UNIX はなぜいいか．もっとも大きいのは，利用者の頭の働きに逆らわず，むしろそれを助長する，という点である．ああやって，こうやって，それからこうやりたいなあ，と思ったとき，そういってやればそのとおり動く．

なぜ UNIX の開発者たちはそういう，利用者の頭の働きをうまくとらえたシステムを作ることができたのか？ Thompson らは文献1)で，こう述べている．

「多分逆説的なことというべきだろうが，UNIX システムの成功は，それがいかなる前もって定められた設計目標に合わせて設計されたものでもない，という事実によるところが大きい．その第1版が書かれたのは，著者の一人（Thompson）が手元の計算設備に不満を覚え，あまり使われていない PDP-7 を見つけ出し，もっと使い心地のよい環境を作り出そうと乗り出した，といういきさつによる．…(中略)…われわれは誰かほかの人の要求仕様を満たすという必要に迫られたことは一度もなかった．われわれはこの自由を，ありがたいことであったと感じている．」

このように UNIX は，構想，要求定義，仕様化，設計，…という通常のソフトウェア製造手法に則って計画的に生産されたソフトウェアではなかった．むしろある意味で，自然に生まれてきたソフトウェアであった．そのことが開発者たちの才能および彼らを囲む利用者群からの友好的な助言とあいまって，利用者の思考の流れにすっと沿う，きめのこまかいソフトウェアを作り上げるもととなった．

さて，そういう実に大好きなソフトウェアのマイナス面を述べることが筆者に与えられた責務である．つまり，求められているのは「悪魔の代弁人」の役割である．ただし悪魔の代弁人とは，たとえばシステムの評価のために会議を開いたとき，わざと否定面について発言する役割を与えられた参加者のことをいい，見落としを防ぐ効用がある（元来はカトリック教会の，聖人を決める手続きに起源を持つアイディアであるという）．心理的にはつらい役柄である．

重ねていうが，UNIX はすばらしいオペレーティング・システムである．ここでいう「悪口」は，それはそれとして，その先へ行くためにあえてする発言である．その点はぜひ誤解のないようにお願いする．

UNIX 批判の論文として，最近では Blair ら[2]のものがある．その中に次のような言葉がある．

「だが昨今見られるわれもわれもと尻馬に乗る風潮は，UNIX オペレーティング・システムとその思想への多大の投資を呼び，ひいてはその分野での学問の進歩に足かせをはめるおそれがある．」

「事情は上級プログラミング言語のはじまりのころと似ている．当時産業界では，FORTRAN や COBOL のような初期の言語が，ほとんどあらゆるところで使われた．プログラミング言語が大きく進歩したあとも，その大きな投資が変化への強い抵抗要因となった．」

筆者も Blair らと同じことを感じている．あえて悪魔の代弁人を勤めることによって，せっかくのすばらしいシステムが，技術の進歩を遅らせるもととならないための一助としたい．ここで述べることの一部は彼らの論点と重なっているが，本文では時代的背景ということを重視する．

2. 基本的設計方針について—その世代論的位置づけ

2.1 ソフトウェア的世代

計算機システムについて世代というとき，それは

† UNIX-Its Limitations and the Future by Izumi KIMURA (Dept. of Information Science, Tokyo Institute of Technology).

†† 東京工業大学理学部

表-1 計算機システム史の世代論的な流れ*

世代	ソフトウェア的思想	システム例	初出年代	特性的時定数
1	記号コーディングとサブルーチンライブラリ	EDSAC	1949	1週間
2	問題向き言語	FORTRAN I	1957	1時間
3	TSS	CTSS	1962	1秒〜数十秒
3.5	ソフトウェア工具	UNIX	1969	1〜数秒
4	パーソナルコンピューティング	Alto	1973	0.1 秒
5	論理型プログラミング	?	?	?

* A view of the generations of computer systems.

ハードウェア的世代を指しているのがふつうである．第1世代は真空管，第2世代はトランジスタ，第3世代は IC，…という具合である．確かにそういういいかたにも，第3世代あたりまではなるほどと思わせるところがあるが，世代が第4，第5と進むにつれて，どうも変な感じがしてくることは否定できない．時代が下がれば下がるほど，計算機をどういう素子で実現するかもさることながら，むしろソフトウェア的設計思想の重要性が増してくるからである．

計算機システム史の世代論的な流れを，そういうソフトウェア的観点から整理して世代わけしてみると，表-1 のようになる．

たとえば，ソフトウェアにおける最初の一里塚が EDSAC における記号コーディングとサブルーチンライブラリに求められることには異論はあるまい．そこで表-1 では，それを第1世代を代表する思想としてリストアップしている．その初出年代を 1949 年としたのは，EDSAC のお披露目が 1949 年6月になされたことによる[3]．ここではソフトウェア的世代とハードウェア的世代は一致している．事実 EDSAC は，真空管式の計算機であった．

第2世代以下についても，考えかたは同様である．なお Alto は，論文発表は 1979 年まで遅れたが 1973 年に完成してそれ以来実際に使われていた．そこで，1973 年を初出年代としてある．

表-1 にはないが ENIAC は，ソフトウェア的に見れば第0世代のシステムということになろう．それは真空管式の計算機であったから，ハードウェア的には第1世代ということになるが，ソフトウェア的にはソフトウェアという概念そのものが当時はまだなかったと考えられる以上，同じ真空管で作られていたからといって ENIAC を EDSAC と同列に置くことはできない．

さてこうした見かたからすれば UNIX は，表-1 にも示したように第3.5世代のシステムであるといってよい．実際，UNIX は基本的には TSS でありながら，ソフトウェア工具の思想を正面に据えたという点でそれ以前の TSS (たとえば CTSS) とは一線を画している．一方 UNIX と CTSS の差は，EDSAC と FORTRAN I の差，および FORTRAN I と CTSS との差と比べれば，明らかに小さい．

2.2 なぜ3.5世代か？

いま UNIX と CTSS の差は小さいといった．その点については少々補足説明が必要かも知れない．そこで表-1 には各世代のシステムの特性的時定数（利用者とシステムのやりとりにおける待ち時間の典型値）を付記した．すなわち EDSAC 時代の計算機利用者は，たとえば1週間前に開かれる割り当て会議に出て1時間なり30分なりの計算時間を予約する，というのが常識であった（必ずしも EDSAC における歴史的事実がどうであったかではなく，むしろ当時の時代的常識をいえば，そういうことであった）．これに対し FORTRAN I 時代の利用者は，計算センターにカードデックを提出しておけば，1〜2時間後にはラインプリンタに打ち出された計算結果を受け取ることができた．また CTSS の時代の到来とともに，簡単な仕事であれば指令を打ち込んでから数秒とか数十秒とか経てば結果が帰ってくるようになった．

そして次の大変化をもたらしたのは Alto であった．Alto では画面上のカーソルがマウスの動きに追従するようになった．ということはカーソルを，マウスに追従しているかのように見せるのに十分なだけの速さで動かしてやらなければならない，ということであった．それには人の目の追従能力と同程度の反応速度が必要であった．それを表-1 では，ぎりぎりの線ということで 0.1 秒とした．

こうしてみると，表-1 の第1，2，3，4世代の間には，順にほぼ同程度の時定数比があることがわかる．実際，1週間と1時間の比は 168 対 1 である．ところが1時間の 168 分の 1 は 21 秒，そのまた 168 分の 1 は 0.13 秒に当たる．つじつまはよく合っているといってよいであろう．そしてこのことから，UNIX が

3.5 世代だといういいかたには，ある程度の合理性があるといえよう．

UNIX の場合，額面上の時定数は CTSS の，1秒ないし数十秒という値からそう大きくかけ離れてはいない．短くなったといっても，せいぜい1桁程度の短縮である．だからこの，時定数に基づく考えにおいて UNIX と CTSS の間に有意差を認め得るかどうかは一応問題である．だが UNIX では，同じ時定数とはいっても，一般に1回の指令呼び出しによって処理できる仕事の大きさが大きくなっている．したがってそこに 0.5 世代の違いを認めることは決して不当ではあるまい．

余談ながら以上の話を強引に外挿すると，第5世代の時定数は 0.75 ミリ秒，第6世代の時定数は 4.5 マイクロ秒ということになる．システムを使う人の方の情報処理能力に限りがあるから，こういう単純な外挿がずっとやっていけるとは考えにくい．ハードウェアの今後の進歩による利得は，一部は時定数でなく，たとえば画面の分解能の方を改善するのに使われ，さらに余った部分は人がじかにコントロールしきれない部分を計算機の方で「よきにはからう」ためにも使われる，ということになるのかも知れない．そしてそうなれば，計算機がした当て推量がはずれる率を考えに入れなければならないことになるであろう．以上の話はあくまで UNIX の時代的背景を理解するための一つの発見的議論にすぎないので，あまり深読みはしていただかない方がよいと思う．

この，世代当たり 168 倍の違いはハードウェアの進歩なくして生じ得なかったものであり，それが素子の進歩と連動したものであったことは事実である．だが上記の時定数に関する議論はむしろ UNIX は3か 3.5 か4かという問題をはっきりさせるためにしたものである．むしろ表-1 の趣旨は，世代間のコントラストをそのハードウェアの進歩によって与えられた新しい可能性をどこに生かしたか，というソフトウェア的側面に見出そうというところにある．

2.3 工具ソフトウェア層の重視

さて以上要するに，UNIX は 3.5 世代のシステムであり，第4世代のシステムでも第5世代のシステムでもない．時代が進めば，UNIX の寿命が尽きるときはきっとやってくる．そのことの一つのあらわれは，UNIX の土台の部分の作りかたに見られる．ここで土台とは，次のような意味合いである．

基本ソフトウェアを内部に立ち入って眺めてみると，少なくとも三つの階層が見出される．それを仮に①基底ソフトウェア層，②認知ソフトウェア層，および ③工具ソフトウェア層と呼んでみることにしよう．**基底ソフトウェア層とは**，基本ソフトウェアにおいて，たとえばプロセスの管理といった，いわば地盤工事に当たる部分をいう．その層を外側から見たとき，それは現在の技術水準ではマイクロ秒（またはそれ以下）を単位とする世界である．次に**認知ソフトウェア層**とはソフトウェアの中の，人の認知的能力に即応する部分である．たとえばマウスつきのインタフェースにおける画面管理ルーチンなどがこれに当たる．これはミリ秒単位の世界である．認知ソフトウェア層は基底ソフトウェア層の上に乗せる形で作られるのが常識である．最後に**工具ソフトウェア層**とは，以上二つの階層を素材として，利用者が実際に仕事をしてゆく上での基礎的工具類を作り出す階層であって，ソートマージはその典型例である．このほかテキストエディタも，古典的なテキストエディタは工具ソフトウェア層に属する．これは秒の世界である．実際，利用者がソフトウェア工具を使いこなしてゆく上で必要とする思考時間は，1ステップ当たりにすればその程度のものである．そのぐらいは考えなければ計算機にさせるためのまとまった仕事を思いつくことはできない．

さて UNIX は，工具ソフトウェア層に光を当てたシステムである．認知ソフトウェア層はより軽く扱われている．それは，基本的にはいわゆるテレタイプ・インタフェースの延長上にあり，たとえばマウスなどは出発点では想定されていなかった．その不十分さは利用者が空想力で補う建て前である．

UNIX の基底ソフトウェア層は，そこを見越して大ざっぱに作られている．たとえばプロセスの切り換えは，比較的ゆっくりした割合でしか起こせない．事実このことは比較的よく知られていて，「UNIX はリアルタイム処理に向いていない．」というような表現で語られることが多い．また事実このことが，現在わが国で売られているパソコン向きの UNIX システムに，トップクラスのワープロソフトウェアに肩を並べるようなものは用意されていない，という残念な状況の一つの根源にもなっている．

ここで注意しなければいけないのは，だから何かをつけ加えてリアルタイム処理にも使えるようにしよう，というのは基本的には邪道だ，ということである．UNIX において基底ソフトウェア層がそのように作ってあるのは，全体のバランスを考えてのことだ

からである．プロセスの切り換えがゆっくりでよければ，関連部分の作りがずっと楽になり，ひいてはシステム全体としての効率を高めることができる．そこをそうしないで，もっと速い切り換えができるようにとがんばったとすれば，ほかの部分の作りも変えなければならないことになる．そうなれば UNIX は，もはや UNIX でなくなってしまうおそれがある．

このように，もともとそういう風にできている以上，UNIX を第4世代的な使いかたで使おうとすれば，(そういうことは現在ずいぶんあちこちで試みられているが) 多かれ少なかれ無理なことになる．まして，第5世代的（それが具体的にどんなものになるにせよ）な使いかたをしようとすれば，カードリーダ，ラインプリンタつきの第2世代の計算機に何か細工を施して Alto のまねをさせようとするのと同程度の無理を犯すことになるであろう．UNIX のよさはそれを第 3.5 世代らしく使ったときに，もっとも生きてくるのである．

2.4 一応の解答と模範的な解答

では，第5世代がうんぬんされる今日，なぜ3.5世代などという古いものが問題なのか．その答えはこうである．ソフトウェアの問題は，一応解けたと模範的に解けたの間の懸隔が大きい．UNIX は 3.5 世代の問題を模範的に解いた．そうであればこそ，あれほどの人気を集めた．第5世代の計算機システムの問題は，まだその実体すら，どんなものであるか十分明確になっているとはいえない．少なくとも万人のイメージが一致する，というところまではきていない．その実体はいずれは明らかになり，それと並行して解答が得られてゆくことになるだろうが，ただし最初の解答はまず間違いなく「一応解けた」という段階のものにとどまるであろう．それは，そのようにして一応解けたあと，多分技術のフロントエンドが2世代ぐらい進んだあとで，もう一度模範的に解きなおされなければならないことになるであろう．

したがって UNIX は，第5世代以前に，いずれまず第4世代の模範的システムによって，置き換えられなければならない．そのための道筋はどのようなものであるべきだろうか？

UNIX があまりにも模範的であるために，そこから出発して膏薬張りをやろうという誘惑はきわめて強い．だが歴史の告げるところによれば，膏薬張りはソフトウェアの世代交代にとって，致命的な麻薬である．この点については第4章で再論する．

3. その他の諸問題

以上では，UNIX の基本思想についてその位置づけを述べた．UNIX のその面での限界は，いわば戦略的なものであって，少々あがいてもどうなるようなものではない．だが UNIX には，それ以外にも問題点がある．それは端的にいえば UNIX が最初に開発されたときに開発者が予測していなかった環境条件ゆえに生じた，戦術的な問題である．それらも少々の膏薬張りで片付くようなものではないが，世代的限界という前章の問題点に比べれば根が浅く，膏薬張りではだめにしても仕立て直しによってある程度解決する可能性がある．この章ではその種の問題点について手みじかに述べる．

3.1 ぜいたくになった利用者

UNIX は，はじめは作者の身辺で使われていた．利用者と開発者は原則として顔見知りであった．のちには評判が立ってシステムがあちこちに配られるようになり，顔見知りばかりとまでは行かないようになっていったが，それでもまだ不特定多数というのではなかった．また彼らの多くは，高度に知的な利用者であった．だから当時の利用者たちは，システムに注文をつけるときも，開発者の立場をある程度心得た上で注文をつけたに違いない．

また当時は世間一般の計算機システムのできが，今日ほどよくなかった．相当木で鼻をくくったような利用者インタフェースでも，文句をいわずに使ってもらえた．そういうものと比べれば UNIX のインタフェースは，圧倒的にぜいたくな部類に属していた．

だが UNIX は，あまりにもよくできていたために，ついには不特定多数の手に届くことになった．また世間一般の利用者インタフェースが，時代が進むとともに少しずつ改善されていった．そのため UNIX も，もっとぜいたくにならなければ大手を振りにくい立場になってきた．

そこでいろいろ結構な機能が追加された．事実幸いなことに UNIX は，機能をあとからつけ加えやすいようにできていた．またそのための，手触りのよい枠組みを提供していた．一般の計算機システムではそういうことをすると矛盾が生じ勝ちであるが，UNIX の場合には基本設計のよさゆえに，あまり矛盾が生じなかった．

だが，よくできた枠組みも，あまり「重い」ものを乗せればこわれる．こわれるまでは行かなくても，な

んとなく軽快さが欠けてくる．そういう，少々重すぎるものの典型例として C-shell がある．

C-shell はもとの（たとえば UNIX 第6版の）shell に比べればずいぶんぜいたくにできている．たとえばヒストリー機構（過去に入力した指令を参照，編集して新しい指令を作り出す機能）などは大変結構なものである．あるとないではずいぶん使い勝手が違う．

だが，そういうふうなぜいたくな機能がいろいろとつけ加えられたために，C-shell にはところどころ具合の悪いところができた．典型例は，記法

~/…

および

~利用者名/…

に見られる．これらの記法は，もとの shell にはなく，C-shell に追加されたものであって，それぞれ利用者自身のホームディレクトリ，および利用者名で指定された（他人の）ホームディレクトリへの参照をあらわし，非常に便利なものである．しかしこれらは基本的には C-shell にしか知られていない約束ごとであるので，よく混乱が起きる．たとえば make はこの約束を知らないので，(make 自体を書きなおさないかぎり)「そんな『~』などという文字ではじまるディレクトリ名はないぞ．」といって怒り出す．

明らかにこれは，組み込むとすればファイルのオープン処理ルーチンに組み込むべき機能である．そのようにすれば上述のような片手落ちは生じない．だがそうとすると，UNIX の現在の作りかたではシステムの中核部に手入れをすることが必要になる．それはおおごとである．ことに他人のホームディレクトリが指定できるためには，中核部に利用者登録簿のありかを教えておく必要があるという点が困る．利用者登録簿は /etc/passwd という名のごく普通のファイルに入っている．中核部にそんな本質的でないことを教えるのは，よきシステム設計の精神に反する．要するにはじめに想定してなかったぜいたくさが，矛盾の種になっているわけである．

3.2 初心の利用者

利用者が身内ばかりではなくなったことからのもう一つの帰結は，ものを知らない初心の利用者がふえてきたことである．そのため，これまでどうということもなかった問題点が，種々表面化してきた．たとえばマニュアルやヘルプメッセージが，これまでのようなことでは間に合いにくくなっている．

利用者が知的な身内ばかりであったうちは，箇条書き風のマニュアルの方が，世間の商用のシステムによくあるお話風にべったりと書かれた分厚いマニュアルよりむしろ喜ばれた．事実 UNIX は，原理を知ってしまえば，あとは多分こうだろうと思って使うとたいがいの場合に当たる，という好ましい性質をもっているので，余計そうだった．だが初心者は，原理を知るまでに至っていないことが多いから，それでは間に合わなくなった．

ヘルプメッセージも同様であって，昔は指令名を与えるとマニュアルの，その指令のことが書いてある部分を打ち出す，というだけのもの（man 指令）だけで結構間に合っていたのであるが，最近はそれでは利用者に満足してもらえないようになってきている．

利用者のレベルアップが必要であることは確かであるし，あまりそこのところをべたべたさせてしまったら，UNIX が UNIX らしい軽快さを失うおそれも大きいが，ともかくもう少し何とかする必要があることは明らかである．

そしてそれはマニュアルやヘルプメッセージだけを工夫しただけで全部解決するとは限らないからやっかいである．たとえば UNIX は，システム側で入出力のバッファリングをおこない，バッファの内容と外部記憶装置の同期は要求があったとき，またはある一定の時間が経ったときにとるというのが伝統である．またそのことが，UNIX の使い勝手のよさに大きく貢献している．システムの応答時間を短縮するとともに，間接的には利用者を，バッファの管理に関するわずらわしい考慮から開放している．

その代わり UNIX では，システムを止める前に sync システムコールと称するものを発して，外部記憶装置とシステムのバッファの間の同期を強制的にとってやる必要がある．もし正規のシステム停止手順を経ないで電源を切ったりすると，ファイルシステムがめちゃめちゃになる．また停電があったりすれば，めちゃめちゃにしようと思わないでもそうなってしまう．これは初心者にとって，おそろしいことである．熟練者ならそういうときでも，何とか手当てができるものだが，その手当てのこつをマニュアルやヘルプメッセージの形で初心の利用者にもわかるように提供することはまず不可能であろう．最近は，応答時間を多少犠牲にして，ファイルシステムをより丈夫にする試みもおこなわれているが，それにも限度がある．本当に初心者にとっても大丈夫な UNIX を作ろうと思えば，無停電装置でも組み込むしかないかも知れない．

3.3 新種の利用者

UNIX がはじめて作られたとき，日本人利用者は開発者の眼中になかった．身内には日本語しかしゃべれない日本人はいなかったはずであるから，それは当然であった．

その結果，今日日本に住んでいる日本人の目から見れば見落としとしか見えないものがいくつか生じた．その一つは文字コードという問題である．

原 UNIX は，文字は7ビットで表現できると堅く「信じて」いる．そしてその信念が，入出力のドライバはもとより，たとえばテキストエディタの中などにも，深くしみこんでいる．バイトの8ビット目が文字コードの一部としてではなく，何か別の目的に使われていたりする．

したがってそこに日本語を乗せようとすると大変なことになる．この問題は，現在組織力と人海戦術によって解決に向かいつつあるようであるが，そもそも組織力が必要になるというのは，UNIX の精神からすればあってはならないはずのことである．

もう一つのよく似た問題は，利用者名の長さという話である．現在それは8文字までということになっている．日本人の場合，たとえば姓のローマ字表記を使いたいところだが，それが必ずしも8文字に納まらない．高橋のようなありふれた名前でも，takahashi とは打てない．UNIX のようなよく考えられたシステムなら，それを 16 文字にすることぐらい何でもないかと思うととんでもない．実にいろいろなところにからみがあって，簡単な手なおしはできないもののようである．利用者はどうせ身内なのだから利用者名は頭文字を取って klt とか bwk とかいったものでいいではないか，というのが開発者の意図だったと思われるが，日本人にはミドルネームというものがないので，頭文字では，ことに少し利用者集団が大きくなると頻繁に衝突が起こってしまってだめである．これは初期の意志決定が10年以上あとになって裏目に出た実例，といえる．

3.4 ハードウェア的環境の進化

初期の UNIX が載っていた計算機は，いまならそれと同程度の能力を持つ計算機を，どうかすると高校生でも持っているかも知れない，という程度のものであった．そういうハードウェア的環境の進化は，UNIX のある部分を大きく時代遅れにしている．

たとえばビットマップ・ディスプレイという問題がある．それを UNIX の本来の枠組みの中で扱おうとすれば，ビットマップ記憶装置を特殊ファイルとして扱う，というのが自然であろう．しかしもしそうとすると，ディスプレイに対して何かするごとに，ファイルの読み書きに関する一定のおまじないが必要となる．ビットマップ・ディスプレイはことさらすばやく動く必要のある装置であり，上記のおまじないは，そのすばやさに対する耐えがたい負荷となるおそれがある．なおこれは 2.3 節で述べた，工具ソフトウェア層対認知ソフトウェア層という話の，一つのあらわれでもある．

3.5 ソフトウェア的環境の進化

UNIX では，現在のところ C[4] が標準言語になっている．第6版以降の UNIX では OS 自体，中核部の一部を除けばCで書かれており，それが UNIX の機種間通用力に大きく貢献している．

だがCは原始的な言語である．アセンブラよりはどれだけ安全かわからないが，おそろしいことを起こそうと思えばいくらでも起こせる言語である．たとえば配列の範囲外をさわって，とんでもない書き換えをやってしまうようなことが，FORTRAN によるのとそれほど違わないぐらい容易にできてしまう．少なくともそういうことは，言語の構文上は禁止されていない．事実オペレーティングシステムは，どこかにハードウェアをじかにいじる部分を含んでいないではすまされない．そういうおそろしいことができる言語でなければ，オペレーティングシステムを書ききることはできない．だがそれが必要なときにできるのと，至るところでできるのは別問題である．

Cは FORTRAN と比べれば，すっきりした言語である．実際，第6版あたりの UNIX では，Cはちょうどぴったりした言語であった．だがCは，たとえば Berkeley 版の UNIX には，原始的すぎる．

UNIX オペレーティングシステムのような，大きなプログラムを作るときには，プログラムのあちこちから操作される共通データを手際よく記述できないと苦しい．そのためにはデータ抽象，またはそれに対応する機構がほしい．

そこでそういうときCでは，include ファイルを使う．すなわち，共通データの定義を適当なファイルの形にまとめておき，プログラム上のそれを必要とする部分にはそれを取り込めという意味の文（#include 文）を書くようにする．

このやりかたは，同じ UNIX でも第6版程度の規模のものであればまずまずうまくゆくが，その後の版

では苦しさが増してきている．include ファイルだけでもおそろしい分量になり，とうてい一人の頭の中には納まりきれない．

UNIX がはじめてあらわれてからこのかた十数年の間に，世間のソフトウェア的な環境はずっと進歩した．たとえばオブジェクト指向言語が普及した．いまから作りなおすならCでなく，何かそういったものを使うのが筋だろう．だがC版の UNIX に対する投資はいまや莫大な額にのぼっており，そのいまから作りなおすという可能性は，いうはたやすく実行はむずかしいものとなっている．

3·6 社会的環境

はじめにいったように，UNIX の開発者である Thompson たちは，他人が設定した設計目標にわずらわされることなしに自分たちが使いたいシステムを作るというぜいたくをすることができた．それが彼らにとって，多くの利用者が使いたいと思う UNIX というソフトウェアを作ることを可能にするもととなった．だがそれは「いいご身分」ということでもあった．いいご身分ゆえに，彼らの作品には「いい気なところ」，つまり商品として売り出すつもりならちょっとどうかな，といいたくなるようなところがあった．

その典型例は，(これは UNIX の UNIX らしいところでもあるのだが) マニュアルに置かれた Bugs (虫) というセクションに見られる．そこにはコマンド，ライブラリ・サブルーチンなどに残っている既知の虫，拡張を要するところなどがあからさまに記されている．元来 UNIX は仲間うちで使うために作られたオペレーティング・システムであったから，そういうセクションが置かれているのは至って自然なことであった．仲間たちに使わせるためのソフトウェアならば，利用者の方でちょっと気をつければ簡単によけて通れるような虫は，根絶されるまで「出荷」を差し控えるなどという必要はない．仲間たちは早く使いたいと思って心待ちにしているのだから，早く使わせてやらない法はない．もちろん彼らがそういった，いわゆる「虫」に引っかかって悩んだりしては困るので，問題点はあらかじめあからさまに書きつけておく必要がある．だが当面はそれで十分である．またそれは開発者自身にとっても，あとで手が空いたとき手を戻すための心おぼえとして役立つ．

さらにふつうの意味での虫ではないが，本当はつけておくと「かっこいい」と思える機能についても同じ扱いをすることができる．「ほんとうはこうしたいんだけれど，まだやってないのさ．」とだけ書いておく．実はずっとほったらかしておく，というのでもよい．ちゃんとなおそうとすると大騒動で，その手なおしは退屈きわまる仕事で，しかも手なおしをしてみたところでそれほど世間が明かるくなるわけでもない，というような小さな不具合を，知っていながらほったらかすというのは，相手が友好的な利用者ばかりであれば，むしろあるべき姿である．

だがソフトウェアが商品になったときマニュアルに平気で Bugs などという項目を置いておけるかどうかはちょっと問題である．「社の恥だ．」というような声が重役会あたりからあがるかも知れない．それかあらぬか国産のパソコン用 UNIX には，マニュアルの中でそこを「注意」としているものがある．「注意」ではニュアンスがずいぶん違う．開発者側の無謬性を主張し，責任を (間違って使ったら間違った方がバカだ，よく気をつけろ，といって) 利用者に押しつけている，と取られても仕方がないようなところがある．

この点に限っていえば，筆者はむしろ「いいご身分」の方を大事にしておいた方が世のため人のため，という気がする．過去においてまじめなソフトウェア生産は，とかく少々きまじめすぎた．そのためにいわゆる過冷却の誤りを犯す事態が稀でなかった．UNIX の商業的成功は，その点を是正するよいチャンスをもたらしたのではないか？ むしろ Bugs は Bugs であっていいのだ．UNIX をその点で商品らしくしようとするのは，おそらく誤りだ．ここでソフトウェアと開発者と利用者の関係について考えなおしてみることこそ，健全な道なのだと思う．

とはいうものの当面の問題としては，UNIX の一層の普及につれて，それを取り囲む肩肘を張らない文化があちこちで摩擦を起こすようになることは疑いがない．これは UNIX の欠点とか限界とかということであるよりは，世間がまだ十分開けていないことのあらわれであり，ここで触れるのはお門違いのきらいもあるが，重大な問題であるから特に記して読者の注意を喚起したい．

4. 今後の進路

このように UNIX は，それ自体すばらしいシステムではあるが，現在のままの姿でずっと生き続けるべきものではない．そう遠くない将来，UNIX の真の精神を受け継ぐとともに，その時代的背景ゆえの限界を克服した新しいシステムがあらわれて，それに取っ

て代わることが望ましい.

あるべき新システムは, UNIX に小さな手なおしを積み重ねてゆくことによって, 徐々に得られるようなものではないであろう. むしろ発想を新たにして書きおろすことが必要だろう. 少なくともその発端の部分では, 作ろうとして作ったのではうまくゆかないだろう. もとの UNIX がそうであったように, 少数のすぐれた人々が, 友好的な利用者群に取り囲まれて, 試行錯誤の繰り返しを経てじわじわと作り上げるという, ソフトウェア開発工業化の必要性が広く説かれている昨今の流れとは一見逆のやりかたが不可欠だろう.

またそのような試みは, 1箇所で行われたのでは不十分だろう. 生物の進化がそうであるように, あちこちでおこなわれ, その多くは流産に至るような, 一見むだの多いプロセスを経て, はじめて達成されることになろう. わっと人が集まって, わっしょいわっしょいとおみこしを担いだからといって, いいシステムができるとは限らない. UNIX 自体がそうであったように, 世間の隅の方で地味に, 自分の信ずるところに誠実であるような人々が作りだしたシステムこそ, 本命である可能性が高い. 読者の多くにとって気に入らない結論であるかも知れない. しかしこれは, どう見ても避けがたい結論であるように感じられる.

参 考 文 献

1) Thompson, K. L. and Ritchie, D. M.: The UNIX Time-Sharing System, Bell System Technical Journal, Vol. 57, No. 6, Part 2, pp. 1905-1929 (July-August 1978).
2) Blair, G. S., Malone, J. R. and Mariani, J. A.: A Critique of UNIX, Software-Practice and Experience, Vol. 15, No. 12, pp. 1125-1139 (Dec. 1985).
3) Randell, B., ed.: The Origins of Digital Computers, Second Edition, Springer Verlag, Berlin (1975).
4) Kernighan, B. W. and Ritchie, D. M.: The C Programming Language, Prentice-Hall, Englewood Cliffs, N. J. (1978). 石田晴久訳, プログラミング言語C, 共立出版, 東京 (1981).

(昭和61年10月31日受付)