Windows Subsystem for Linux, Linux を食べた Windows

千種 菊理　著

2016-08-14　　Nagisa unworks　発行

第 1 章

Windows で Bash が動く!?

1.1 Build 2016

2016 年 3 月 30 日より 3 日間、サンフランシスコにてマイクロソフトの開発者向けイベント **Build 2016** が開催された。アップルの **WWDC** がそうであるように、開発者向けのイベントとは今後搭載される新技術や機能が紹介される、今後を占うに重要なイベントだ。今回の Build では Cortana (Siri に相当する音声インターフェイス) を拡充、各アプリケーションからも積極的に利用できるようにしたり、Skype や Line といったチャット (テキストや音声) に受け答えするアプリケーションをより作りやすくするなど数々の発表があった。

その中でもひときわ歓声が上がったのが **Bash on Ubuntu on Windows10** であった。

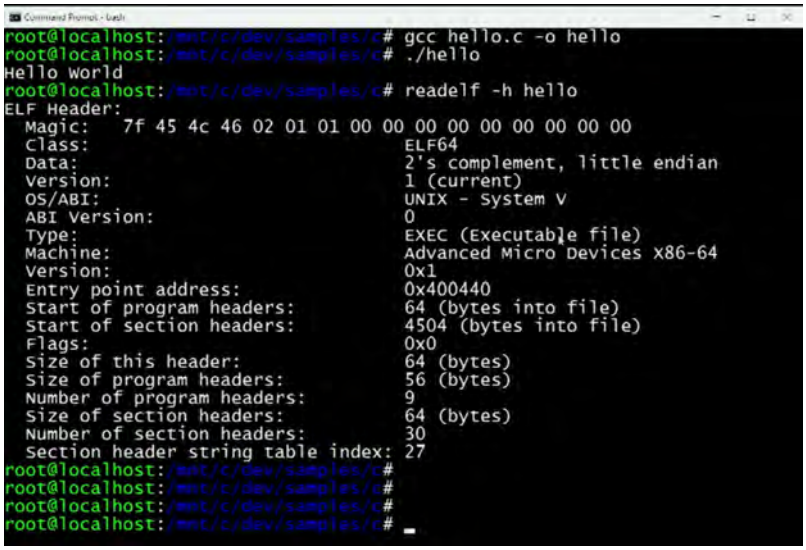

図 1.1　Build 2016 でのデモの一つ

bash とは UNIX/Linux で使われているシェルの 1 つ。その bash やその他 UbuntuLinux のコマンドを Windows10 でそのまま実行するデモ[*1]が行われた。

[*1] https://channel9.msdn.com/Events/Build/2016/P488 など

1.2 何が大事だったのか

bash が動く、それ自体は大した事ではない。bash の Windows への移植版もあれば、cygwin という、UNIX 系ソフトウェアを Windows 上に移植しやすくするミドルウェアと実際に移植されたコマンドライン群のセットも存在しておりよく使われている。また、VMware などの仮想化ソフトウェアを使って Windows 上で Linux を動かすことだって難しくはない。Windows 以外の OS を見れば、たとえば Apple の OS X には UNIX の 1 つである BSD が丸ごと入っており、Mac ユーザは使いたいときにいつでも bash が使える。

重要な点は、この Bash on Windows10 がそのどれでもない、Windows 向けに移植した bash でもなければ仮想化された Linux でもない、**Linux** 向けのバイナリがそのまま **Windows** 上で実行されていると明言されたことだ。言ってみれば「**Windows10** が同時に **Linux** でもある」 この二つの OS の長年の確執を知っていれば、にわか信じがたいことが起こったわけだ。だから衝撃と歓声があがったのだ。

では、Windows はどうやって Linux を「喰って」しまったのだろうか? それは、20 年近く前に同じように異文化である BSD を「喰った」Apple の OS X と何が違うのだろうか?

1.3 マイクロカーネルとパーソナリティ (サブシステム)

今の Windows も OS X もどちらも、**マイクロカーネル**という考え方を基盤においた OS となっている。

OS X の場合は Mach というマイクロカーネルが中枢にあり、OS として本当に必要な機能「プロセス管理」「メモリ管理」「プロセス間通信」の 3 つだけを処理している。

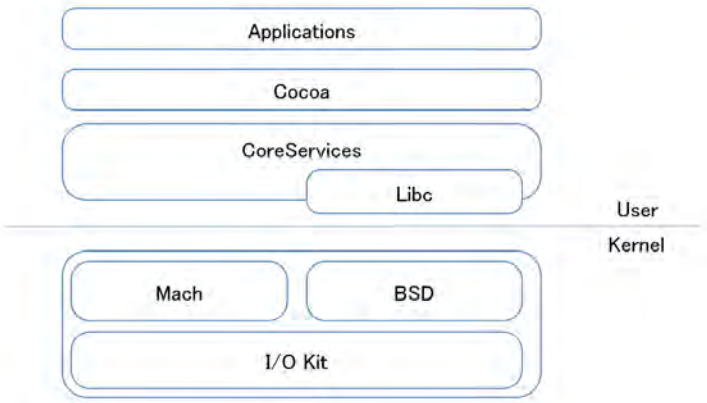

図 1.2 OS X の構造

たとえば Safari という Web ブラウザ (アプリケーション) を起動したとき、必要なメモリを確保して割り当てて (メモリ管理)、定期的に CPU を割り当てて実行させ (プロセス管理)、カット&ペーストなどで他のアプリケーションと連携したり、OS を呼び出したりする (プロセス間通信) は Mach が行っている。それ以外の、例えば TCP/IP でインターネットに接続してウェブサーバからコンテンツを取得したり、ダウンロードしたファイルをディスクに保存したり、その時にアクセス権をチェックして書き込めない場合はエラーを返す、などは Mach カーネルは「やっていない」。ファイルシステムやネットワーク、ユーザアカウントの管理などは「パーソナリティ」という形で、同じカーネル内の BSD カーネルの部分が担っている。さらに、実際のハードウェアにアクセスする部分は I/O Kit と呼ばれるデバイスドライバ群とそのための枠組みが対応している。

60 年代に開始された Mach の開発のそもそもの発端は、今のようにコンピュータが身近に沢山あるような時代が来ることを予想して、並列かつ分散した状態で OS がいかにあるべきか、を追求することだった。当時の UNIX などの OS は様々な機能がつぎはぎで次々追加されており、そうした新たな時代に必要な拡張が困難と思われたからだ。古い OS の轍を踏まぬようにデザインされる中で現れたのがマイクロカーネルだ。曰く、OS の機能にはメモリ管理のように時代を経ても変わらず必須のものもあれば、そうでないものもある。例えばマルチユーザなどがそうだ。 Mac では複数の利用者を定義してログインするのが当たり前だが、よりパーソナルなデバイスである iPhone では利用者は 1 人であり、マルチユーザはあまり考えられていない。カーナビや券売機といった機材になればそもそも特定の利用者がログオンすること自体が不要になる。

こうした場合により必要かが変わる、機能や味付けが変わる部分を「パーソナリティ」という形でモジュール化しておけば、新たな用途、例えばロボット、が生まれたときもパーソナリティだけ差し替えればよく、OS そのものを再開発は不要だ。

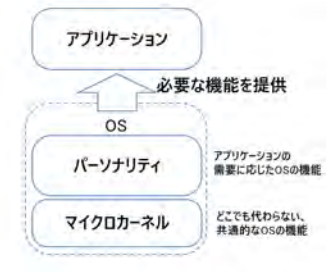

図 1.3　OS として必須の最低限の共通機能と、需要によって代わる機能を分離することで構造を分かりやすくなり、開発期間の短縮、バグの削減等が狙える

この思想のもと開発された Mach は一定の成功を収めた。Mach 2.5 まではパーソナリティと Mach カーネルは構造的には分かれていたが実行状態としては同じ「カーネル」の中に格納されていた。パーソナリティを置き換えたい場合は再ビルトが必要だった。

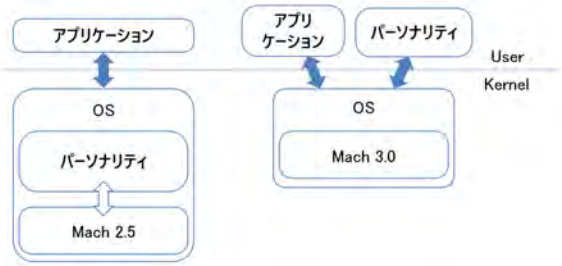

図 1.4　Mach 3.0 では 1 つのカーネルのバイナリからパーソナリティを追い出し、アプリケーションと同じように配置できるようになった

Mach 3.0 ではパーソナリティを完全にカーネルから追い出し、動的に置き換えが可能となった。ここで「パーソナリティを入れ替えればどんな OS にでもなる」、また複数のパーソナリティを同時に実行することで複数の OS のアプルケーションを同時に使えるなどといった夢が生まれてきた。[*2] 90 年代後半、こうしたマルチパーソナリティ OS は明日にでも実現しそうな夢の技術だった。[*3]

ただ、この夢は夢のままの状態が長らく続く。Mach 3.0 は実際には非常に低速だった。パーソナリティを外部に追い出したことで、カーネル内で高速にやりとりされていたパーソナリティと Mach の間のやりとりが、より低速なユーザ空間とカーネル空間のやりとりになってしまい、無視できないほどの性能低下を生んだからだ。

90 年代から 00 年代を通じて、Apple は Mach の中心開発者のひとりであるアビーテバニアンをソフトウェア担当上級副社長として招聘し、Mach をベースとした OS X を開発、成功を収めた。[*4]一方 マイクロソフトは Mach を主導したリチャード・ラシッド教授をマイクロソフト研究所に招く一方、当時から優れた OS 開発者として名高かったデヴィッド・カトラーを雇いあらたなマイクロカーネルベースの OS である WindowsNT を開発させた。WindowsNT は当初サーバ向けの OS だったが、Windows 2000, WindowsXP にてクライアント系の OS でも WindowsNT をベースとするようになり、現在の Windows 8, 10 などにつながっている。

一見 2 つの OS は同じくマイクロカーネルから生まれている。しかし、「bash」を動かすという意味では大きく異なっている。

[*2] 今でもこのマルチパーソナリティのマイクロカーネルを追いかけているのが GNU Hurd とも言える。

[*3] 例えば、IBM が開発していた WorkplaceOS がある。IBM の PC 向け OS だった OS/2 を、CPU に PowerPC を使った CHRP という規格のパーソナルコンピューター向けに移植すると共に、OS/2、同じ PowerPC を使用する Apple の MacOS などを「パーソナリティ」として利用できる、Mac のアプリと DOS 由来のアプリがどちらも使える夢の OS が 2000 年ぐらいにはでるはず、だった。

[*4] OS X の Mach は 3.0 から派生したものだが、わざわざマイクロカーネルとパーソナリティを同じカーネル空間に納めるモノリシック構造にしている。

1.4 パーソナリティと結合された OS X、マルチパーソナリティの Windows

OS X では BSD というパーソナリティはカーネルの中にも存在し、Mach と強く結びついている。CoreService (Carbon) も Cocoa も、BSD とおいうパーソナリティの上にある事は変わりない。今からは信じられないかもしれないが、 90 年代後半から 00 年代初めのアップルは倒産ないしは買収も近いというほど危機に瀕していた会社で、悠長にソフトウェア開発をしている余裕がなかった。また当時の PowerMac は、それなりには高速なものの今ほどメモリも CPU パワーも潤沢ではなかった。新 OS を開発する必要はあったが余裕はなかったのだ。そこで、既に稼働している Mach + BSD の OS をもつ NEXT Software 社を買収しすぐに動く OS を確保しつつ、パーソナリティをカーネルの中に入れて性能を確保させた。GPU を用いたグラフィックなどのチューニングも並行して進めて、10.2 Jaguar では実用速度で稼働し、10.3 Panther 以降はアップルの看板 OS として恥じない機能と性能を持つようになった。

一方、同じく MS-DOS という古い OS に引っ張られた Windows 3.1 に動きを縛られていたマイクロソフト社も、新しい OS の開発の必要性があった。が、こちらは当時からして圧倒的なシェアがあり、潤沢な予算と、古い OS の制限がありつつも十分実用的に動作する Windows95 系の OS もあり、そこまで追い詰められてはいなかった。一方、IBM と共同開発した OS/2 のケリを付ける必要もあり、また米国の政府機関は POSIX 準拠を要望していたのでそれに応える必要もあった。そこで、新規にマイクロカーネルベースの OS を作成し、従来の Windows の API を再編したパーソナリティを乗せて WindowsNT を作ってられる余裕があった。なお、Windows ではパーソナリティの実装をサブシステムと呼ぶため、Win32 サブシステムが正しい名称になる。WindowsNT のリリース時には、Win32 サブシステムだけではなく、、OS/2 サブシステム、POSIX サブシステム[*5] などさまざまなサブシステムを乗せた「マルチパーソナリティ」の OS としてリリースされた。

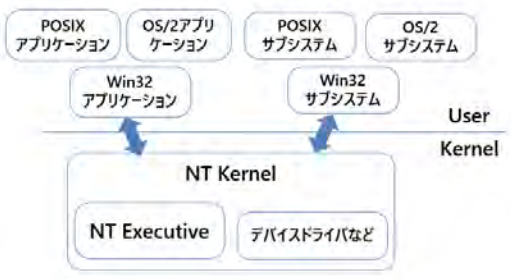

図 1.5　WindowsNT はマルチパーソナリティのマイクロカーネルベースの OS だった

が、Windows 2000, WindowsXP と経るにつれ、全く使われなかった OS/2 サブシステム

[*5] POSIX は米政府により UNIX を定義する規格の 1 つ。10.5 Leopard 以降は OS X も POSIX 準拠の OS である

が削除され、性能確保のために Win32 サブシステムの一部 (GDI というグラフィック処理の中枢) がカーネルの中に組み込まれ、政府機関への入札時の条件として必要なために設けられた POSIX サブシステムは諸々の経緯の後[*6]、 Service For Unix (SFU) という名でオプションとなっていった。WindowsXP の頃には OS X と同じく、ほぼ単一のサブシステムを実行する、カーネルにパーソナリティを混在させて性能を稼いだ OS だったのだ。

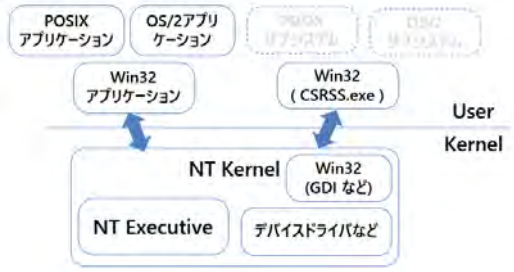

図 1.6　NT4, XP では GDI など Win32 のモジュールがカーネルに組み込まれ高速化された。この構造は Vista で再度改められる

しかし、Windows Vista [*7]で描画システムを改め再び カーネルの外に配置される。ピュアなマイクロカーネル OS に復帰したのだ。

1.5 Windows Service for Linux, パーソナリティとしての Linux

そして今回の「Bash on Windows10」では、Linux カーネルをパーソナリティとした **Windows Subsystem for Linux**（WSL） が実装された。OS X が BSD をパーソナリティとして持つが故に bash がそのまま動くように、「Bash on Windows10」では Linux サブシステム上で bash が動作する。さらに Linux で標準的なバイナリフォーマットである「ELF」にも対応、Linux サブシステムでは通常の Linux にあるバイナリがそのままコピーして動作するようになっている。冒頭の歓声は、このためだった。

この Linux サブシステムは、Win32 サブシステムとは全く独立に動作する。カーネルだけを共有してユーザアカウントやファイルシステムなどは Linux 側の仕組みで動作する。現在の β 版では root アカウントと見なされ動作し、ネットワークについてもまだ未実装な機能がある。ファイルシステムは Linux のファイルシステムがあり /mnt/c/ 以下に C ドライブからのファイルがあるようにマップされている。将来的にはこうした部分の実装も行われ、「Linux サブシステムだけ使っていればただの Linux にしかみえない」ようになるだろう。

[*6] 次章にて説明致します。
[*7] ただ、Vista が遅かったもこれが原因とも言われる

1.6 Windows10 は開発者にとって最高の OS?

インターネット、特にウェブの発展と共に開発者が持つパソコンと言えば MacBook が当たり前になっていた。理由としてはやはり「サーバ環境でよく使われる UNIX の機能を持つ」と同時に「Microsoft Office など業務で必要なデスクトップアプリケーションが動く」を両立している点があるだろう。Windows では UNIX 的な機能がどうしても扱いづらく、Linux では業務アプリケーションが厳しい、その両方を満たすから OS X が選ばれてきた一面がある。また、iOS の開発には X Code が必須のため、使い慣れた Eclips や Visual Studio から移行した開発者もまた多い。

ただ、OS X は BSD と上位レイヤが同じパーソナリティとしてまとまってる以上、それぞれに影響を受けるところがある。ネットワーク周りの下層を見ていけば BSD の「socket」インターフェイスの考え方が見え隠れするように上位レイヤーも BSD の影響を付けつつ、一方で、launchd や ASL など独特の仕組みを持つ BSD レイヤーは、FreeBSD とも NetBSD とも違う、独自の BSD UNIX の世界となっている。これは Lion 以降、顕著になりつつある。El Capitan の rootless なども、古典的な UNIX 使いからは面食らう部分だろう。

今回の Windows の Linux サブシステムは Win32 サブシステムとは異なる、並立するものだ。つまり Windows の世界は Windows の世界、Linux の世界は可能な限りピュアな Linux の世界、と分かれている。OS X に比べ面食らう部分、新たに勉強する部分はより少なくなる。例えば、OS X では OSS の追加ソフトウェアの導入には homebrew などいくつかの OS X 独特のパッケージシステムから選ぶ必要があるが、Linux サブシステムは、現在 Ubuntu Linux を忠実に搭載しており、素の Ubuntu と同じ「apt」で同じモジュールがそのまま追加できる。

一方、Windows で業務アプリケーションが動作する事は言うまでもない。

そう考えると、これまで OS X が独壇場だった開発者向けのデスクトップという分野に Windows10 は強力なライバルになっていくのだろう。

第 2 章

WSL 前夜

Windows Service for Linux の説明の前に、Windows でどのようなサブシステムがあり、特に POSIX や UNIX 系のサポートはどうだったのかをまず見てみる。

2.1 Windows におけるサブシステム

前章で Mach を例に取り、マイクロカーネルをベースとした OS がマイクロカーネルとパーソナリティの組合わせでできていることを説明した。WindowsNT 以降の Windows では、マイクロカーネルに相当する部分が **NT Executive** であり、パーソナリティに相当する部分を**環境サブシステム**と呼んでいる。環境サブシステムとしては Windows のパーソナリティを実現する Win32 サブシステム、OS/2 との互換を実現する OS/2 サブシステム、**POSIX** 互換を実現する POSIX サブシステムおよび **Interix** が存在した。このうち、Win32 サブシステムだけは他の環境サブシステムに対して優先する立場を持ち、Win32 API で独占的に画面描画を行う事ができ、ログオン認証などセキュリティを管轄し、NTFS というファイルシステムを管理しファイル IO を実質的に支配していた。他のサブシステムは、例えば POSIX サブシステムや OS/2 サブシステムはコンソールアプリケーションとして、テキストの入出力を行うアプリケーションとして実装され、これが Win32 が提供するコンソールにより表示されていた。

図 2.1　コンソールアプリケーションは Win32 に存在するコマンドプロンプトウィンドウ (ConHost) を通じて入出力を行う

POSIX でどうしても GUI が使いたい場合は、POSIX サブシステム上で X Window System のアプリケーションを実行し、同じホストの Win32 サブシステム上で **Xming**[*1] など X サーバ を実行して表示させるという方法がとられていた。Windows のアプリケーションが自由にウィンドウを作り、GUI を提供できるのとは雲泥の差だった。

■コラム: Win64 サブシステム?

日本語版の Wikipedia などでは、64bit Windows における Win64 API を処理する環境サブシステムを「Win64 サブシステム」と記載している。しかし、少なくともマイクロソフト社の Web サイトに、「Win64 subsystem」という記述は存在しない。英語版の Wikipedia でもそのような記述はない。ググってヒットするのは WoW64 の説明ぐらいだ。Win64 が開発された Windows2000 の末期から WindowsXP, Windows Server 2003 の時代にはもはや複数の環境システムを擁するマルチパーソナリティな OS という芸風は廃れており、いちいち環境サブシステムを気にするような時代でなかった、という事情もある。とはいえ、出所不明の用語を使うのも気持ちが悪いので、ここでは Win64 サブシステムという用語は一切使用しない。

2.2 Service for Unix (SFU) と Interix

さて、WindowsNT には POSIX サブシステムは搭載されたが、ls とか cat など、POSIX で規定されたツール群や、ネットワークファイルシステムである NFS のサーバやクライアント、リモート印刷のための LPD のクライアントやサーバは搭載されなかった。これらは Srevice for UNIX (SFU) という名称で別売で販売された。

POSIX API を提供する環境サブシステムとしては POSIX サブシステムともう一つ、Interix というものが存在した。前者は WindowsNT 3.1 から存在するマイクロソフト社謹製の環境サブシステムである 。そもそも、マイクロソフトは XENIX という PC 向け UNIX を販売していた会社でもあり、また SystemV の開発にも関わっていた。このため POSIX の実装もそこまで苦労はしなかったと思われるが、しかし、使った人の話を総合するに、純正の POSIX サブシステムと当時の Service For UNIX はかなりイケてなかった、らしい。[*2]

海外でも事情は同じで、そこでもうちょっとマシな POSIX サブシステムと、Service For UNIX で提供されているような関連ツールがサードパーティーからも販売された。このサード

[*1] Windows 上で動作する、フリーと言えばフリーだが微妙に自由でない X サーバソフトウェア。ソースコードは公開されているが、後にバイナリの入手に寄付が必要になり、また営利組織では有償になったりとした。このため、最新版の 0.7.x ではなく、6.9.0.31 という、寄付や有償化の前にリリースされた最後のバージョンが未だに出回っている。筆者は検証環境の Linux に Oracle をインストールするときに、リモートでインストーラの Java が吐きやがる GUI を操作するのに利用していた。

[*2] 筆者は 20 代の頃に UX/OS (NEC の提供する SystemV の UNIX) でソフトウェア開発をしていたが、C のヘッダファイルのいくつかマイクロソフトのコピーライトを見つけて妙な感動をした記憶がある。当時隣で WindowsNT ベースの開発をしているチームもいたが、さすがに POSIX サブシステムでなんかやろうという変人はいなかったので伝聞でしかない。

パーティー、Softway Systems が実装した環境サブシステムのが Interix だ。最終的に Softway Systems はマイクロソフトに買収される。

Service For Unix 3.0 からはこの Interix をベースに置き換えられている。Service For UNIX 3.5 にて無償化され、WindowsServer 2003 や WindowsXP で使用する事ができた。もっとも、この時期には Windows 上で UNIX 系ツールを使うなら Cygwin を使うのが一般的になっており、あまり使われることはなかった。

後に、LPD や NFS のサーバやクライアントは Windows 側に標準搭載になり、SFU は Service for UNIX Applications (SUA) という名前に改称される。SUA は WindowsServer 2012 までは利用できたが、現在では非推奨となっている。

機能	2008,2008R2	2012	2012R2	2016(TP5現在)
UNIX ベースアプリケーション用サブシステム (SUA)	●	▲非推奨	× (削除)	×
NFSサーバー NFSクライアント	●	●	●	●
NfsShare.exe	●	●	× (削除)	×
NFSv2 サポート	●	●	× (削除)	×
NISサーバーとツール(RSAT)	●	●	× (削除)	×
Telnetサーバー	●	●	× (削除)	×
Telnetクライアント	●	●	●	●
LPDサービス	●	▲非推奨	▲非推奨	▲非推奨

図 2.2　各 WindowsServer での対応状況

2.3　Portable Executable (PE)

では、SFU の上でどの UNIX のバイナリが動いたのだろう? 実は SFU はバイナリ互換ではなく、ソース互換であった。Linux やなんらかの UNIX のコマンドがそのまま動くわけではなく、ソースコードを持ってきて、SFU 向けにビルドしなおすことで動作させていた。

その最大の理由が、**Portable Executable**（PE） である。PE とは Windows における実行バイナリのファイル形式である。[*3]

WindowsNT 以降の Windows は、この PE 形式のファイルを読み込み、実行することができる。逆に言えば、PE しか実行できない。POSIX アプリケーションもバイナリとしては PE にするしかなかったのだ。

一方、Linux は ELF と呼ばれる実行バイナリの形式を使用している。Windows には ELF 形式の実行ファイルを読み込み、メモリ上に展開、実行する能力はないので、Interix サブシステムと同じように Linux サブシステムをただ実装しても、Ubuntu Linux のバイナリをそのまま動かすことはできない訳だ。

[*3] 過去のマイクロソフトの OS, MS-DOS や Windows3.1 などでは COM,EXE と呼ばれる実行ファイル形式を使っていたが、これらは x86 の CPU にひどく依存しており、拡張性が見込めなかった。そこで、AT&T が SystemV という UNIX 向けに定義した COFF 形式を元に、拡張して作ったのが PE 形式だ。以前の COM,EXE に比べ、x86 以外の CPU でも利用でき、柔軟性や拡張性が高いことから、「ポータブル」と称されたのだ。

第 3 章

WSL の実装

さて、では Windows10 で実装された Windows Subsystem for Linux について、どういう仕組みなのかをみてみよう。

3.1 WSL のインストール

Windows10 に 8/2 に公開された Anniversary Update [*1]を適用すると、「Windows の機能の有効化と無効化」に「Windows Subsystem for Linux(Beta)」の項目が現れる。これをチェックすると Linux サブシステムが Windows10 に組み込まれる。

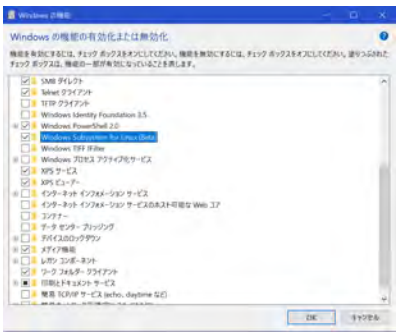

図 3.1　Windows の機能の有効化と無効化

次に、開発者モードを有効にする。これは [設定] アプリケーションの「更新とセキュリティ」の「開発者向け」の項目にある。通常は「Windows ストア アプリ」に選択されており、安全性の高いアプリケーションだけが実行可能なようになっている。WSL はまだ Beta のため、開発者としての処置が必要になる次第だ。[*2]

[*1] 8/2 にリリースされた Windows10 の 2 番目のアップデートで、1607 ないしは Build 14393 と呼ばれている。

[*2] なお、開発者モードを有効にすると、「Windows 開発者モード」というオプション機能がインストールされる。この中に SSH サーバが含まれており、自動的に SSH による接続を受け付けるようになってしまう。

図 3.2　開発者モードを有効にする

スタートメニューに登録された「Bash on Ubuntu on Linux」のアイコンをダブルクルックすると、Bash が起動する。初回起動時には WSL をインストールするか聞いてくるので Yes と答えると、Ubuntu のディストリビューションが丸ごとダウンロードされてきて展開される。これは各ユーザごとになされるので注意してほしい。つまり、同じ Windows10 で 10 個のユーザアカウントがあり、それぞれに bash.exe を実行し WSL をインストールすると、10 倍のディスク領域が取られてしまう。

二回目以降は展開された Ubuntu のディストリビューションをそのまま利用する形で bash が起動する。

図 3.3　Bash on Ubuntu on Windows のアイコン

スタートメニューに登録されたこのアイコンから実行されるコマンドのパスは「\Windows\system32\bash.exe」になっている。しかし、実はこれは bash そのもの、ではない。

この bash.exe は Win32 のアプリケーションであり、WSL を起動するためのコマンドに過ぎない。

図 3.4　bash.exe はあくまでランチャーである

この、bash.exe を起動すると、もしカーネルに WSL がロードされていなければ LXCore.DLL および LXSS.DLL というモジュールが組み込まれる。また、Lx Session Manager(LxssManager) が起動する。

図 3.5　Lx Session Manager もしくは LxssManager

LXSS は pico process (後述) という形で Ubuntu ディストリビューション中の init を実行する。[*3] init が最終的にユーザのシェルとして bash を立ち上げる。なお、このとき /etc/passwd のログインシェルの行は無視されている。

bash の出力はカーネル内のモジュールと LxssManager を経由し、コンソール出力としてコマンドプロンプトウィンドウ (ConHost) に出力される。

[*3] 余談だが、この init は systemd ではなく、Upstart だ。

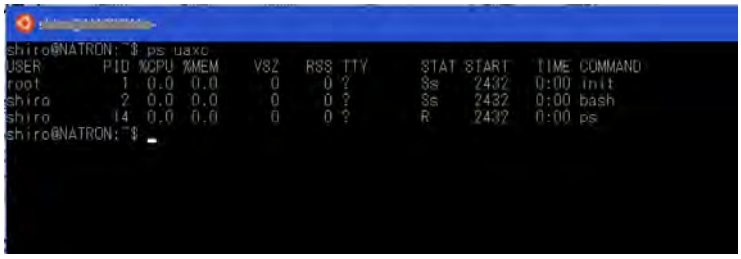

図 3.6　Linux コマンドの実行結果

bash.exe 自体は通常のコンソールアプリケーションのため、ショートカットからではなく、コマンドプロンプトウィンドウを開いてから、bash と実行してもいい。このとき、cmd.exe (または powershell.exe) のカレントディレクトリが「\path\to\dir」だとすると、bash は「/mnt/c/path/to/dir」をカレントディレクトリとして起動する。通常の Windows と WSL は異なるフォルダ階層を持つが、WSL 側からは /mnt を通じて Windows のディレクトリ階層にアクセスができる。

3.2　pico process

WSL のキモが、この pico process を使った ELF バイナリの実行だ。

pico process というものは Microsoft Research で開発された Drawbridge より持ち込まれた、より軽量なプロセスを指す。図 1.15 の右側が通常の Windows でのアプリケーションプロセスだ。

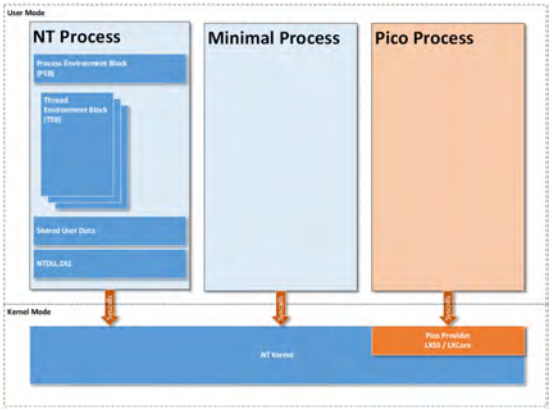

図 3.7　https://blogs.msdn.microsoft.com/wsl/2016/05/23/pico-process-overview/ より

Windows の NT カーネルはアプリケーションの実行時に、アプリケーションが必要とするメモリを確保した後、アプリケーションの実行に必要なモジュールをそのメモリ内にロードする。全てのプロセスで共通の NTDLL.DLL や、アプリケーション間で共有されるユーザの情報、スレッドが実行するのに必要な情報 (TEB) 等々だ。もちろん、アプリケーションのバイナ

リや DLL もこのプロセス内に展開される。アプリケーション実行時にはアプリケーションが必要な情報はメモリに全部用意された状態なのだ。

これは便利だが、言い換えれば、Windows のお仕着せの方式でメモリが利用されてしまっている。何らかの理由でメモリ上の位置を最初から変えておきたい場合、これは不都合が多い。

Windows10 では、Minimal Process と Pico process というこれまでとは違うメモリの使い方をしたプロセスを用意した。

3.2.1 Minimal Process

Minimal Process はメモリ内のお仕着せの準備を一切やめた。プロセスが用意されたときはメモリはすっからかんで用意される。スレッドも用意されないので、このプロセスはほっとくとメモリを食ってるだけで CPU がスケジュールされず、つまり実行されない。それ以前に実行すべきプログラムもメモリ上に展開されていないので CPU が割り当てられても実行することができない。

ただ、環境サブシステムなど特権的なプロセスはこのメモリ内に色々書き込むことができる。適切なコードを割り当て、スレッドを作れば実行もできるようになる。通常の Windows のアプリケーションとは別種のものを実行しつつ、最低限の管理は NT カーネル 側で実施されるわけだ。

Windows10 では、メモリ圧縮や「Device Guard」「Credential Guard」といった仮想化を利用したセキュリティ機能で、この Minimal Process が利用されている。

3.2.2 Pico Process

Minimal Process はメモリはすっからかんでも、特定の手順 (システムコール) で OS を呼び出させば、他のアプリケーションと同じく Windows の OS の機能を呼び出すことができた。

Windows も Linux も、最近の x86-64 (x64) の OS では、CPU のもつ sysenter もしくは syscall という命令を使用する。呼び出したいシステムコールの番号を CPU のレジスタに記録、指定したあと sysenter 命令を実行する。sysenter を実行した瞬間にアプリケーションは停止、各 OS のカーネル側に処理がうつり、CPU のレジスタの中を見てアプリケーションがどのシステムコールを呼ぼうとしたか、どういうデータをわたそうとしたかを確認、処理を行う。ただ、どのシステムコール番号がどの処理に対応しているかとか、そもそも処理の有無や機能が OS ごとに異なっているわけだ。

通常のプロセスでも Minimal Process でも、sysenter 命令が実行された場合の処理は同じ NT カーネルが担う。

一方、Pico Process では、Pico Provider と呼ばれるカーネルのモジュールが sysenter 命令の処理を行う。Windows に代わってお仕置き... ではなく対応を行うわけだ。WSL の場合 LxCore, LXSS のどちらか (多分 LxCore) が Pico Provider になっている。ここには Linux のシステムコールが「そのまんま」実装されている。つまり Windows のシステムコールではなく、Linux のシステムコールの番号として判断され、適切な処理が呼び出される。

もう一度、bash の起動を見てみる。

bash.exe をアイコンから実行すると、これはただのコンソールアプリケーションのため、コマンドプロンプトウィンドウが開き、その中でコマンドが実行される。LxCore, LXSS が必要に応じて NT カーネルに読み込まれ、LxssManager が起動される。

(多分)LxCore が pico process を生成し、(多分)LXSS と LxssManager が協力してファイルシステムにある ELF 形式の init や bash のバイナリを読み込み、すっからかんのプロセスメモリ内に「さも Linux のように」展開を行い、スレッドを作って CPU を割り当てる。通常の命令は CPU によりそのまま実行される。

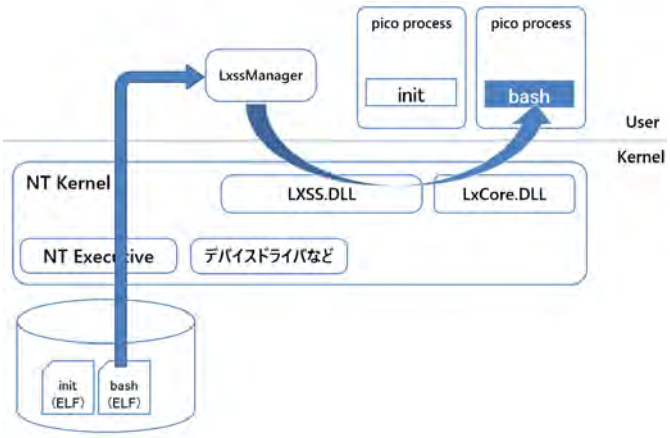

図 3.8　おそらく、LxssManager と LXSS が連携して、空の picoprocess のメモリ空間に ELF バイナリを実行できるように展開しているのだろう

ファイルを読み込んだりテキストを出力を試みると、これは OS の力を借りることになるのでシステムコールが呼ばれる。このとき、sysenter の結果は NT カーネルではなく (多分)LxCore が引き取り、Linux のシステムコールとして処理がなされる。簡単なものなら (多分)LxCore に記述された Linux と同じ処理が行われて、結果が pico process に返される。コンソールへの出力の場合は、LXSS 経由で LxssManager が呼び出され、bash.exe にわたされ、コマンドプロンプトウィンドウに出力がなされる。

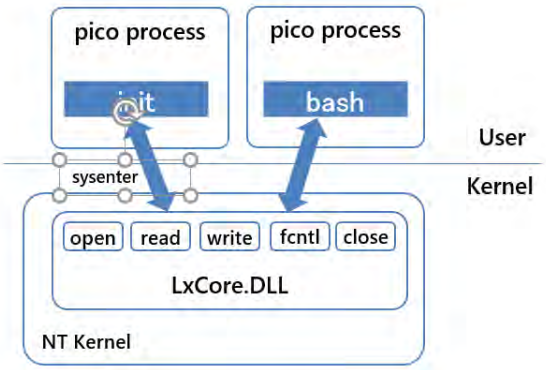

図 3.9　picoprocess のシステムコールは provider である (多分) LxCore が引き受ける。LxCore には Linux カーネル互換の枠組みが組み込まれており、プロセスから見たら同じように動作する

ディスク IO やネットワーク IO の場合は、おそらく、LxCore ないし LXSS から Win32/Win64 API が呼び出され、Windows の該当処理がなされる。

Linux のバイナリは何一つ書き換えられることなく、Linux カーネルの上で実行されていたのと同じように、Windows10 の NT カーネル +LxCore+LXSS の上で実行、できてしまうわけだ。

3.3　Gochas

ディスク IO やネットワーク IO の基本的なものは全く問題なく動作する。そもそも、Ubuntu のアップデートに使用する apt-get などが全く問題なく動作しているのだ。ただ、cgroups など Linux 独特の、少々特殊なシステムコールについては実装されていないか、怪しい挙動をするかであまり信用できない。

Windows からすると Pico Process へは「メモリとプロセスと最低限の OS との通信を提供しているだけ」だ。Win32 プロセステーブルには表示されるが、それ以上ではない。逆に、WSL 側は WSL で実行されている Linux のプロセスは認識できるが、その外で動いている Windows のプロセスは認識していない。一般的な意味でのプロセス間通信はできない。

ユーザアカウントについても Windows のシステムとは独立で用意されている。WSL 側は、/etc/passwd と /etc/shadow を使った古典的な仕組みになっている。この UID と Windows 側とは全く関係がない。bash.exe を利用したときに使用されるアカウントは、Ubuntu のインストール時にセットアップされるが、これはレジストリに記載されている。

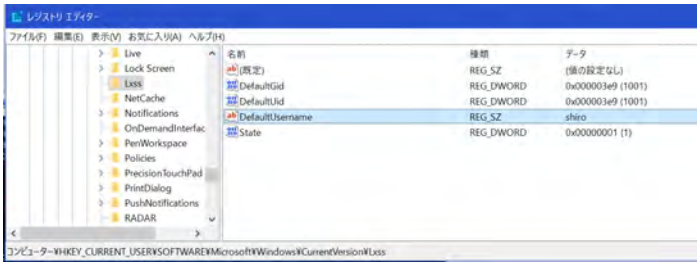

図 3.10　HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Lxss

値を変更したいときは、「 lxrun.exe /setdefaultuser 」で変更が可能だ。[*4]

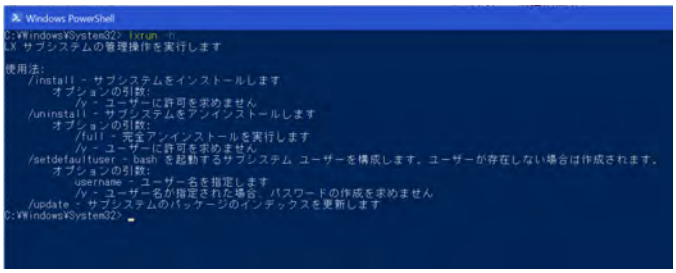

図 3.11　lxrun コマンド

またこのユーザは sudo グループに自動的に参加する。/etc/sudoers で sudo グループに対して sudo コマンドの実行が認められているので Linux のファイルシステムの中では root 権限を行使できる。一方 /mnt/ 以下にマウントされた、Windows 本来のドライブ上のファイルに対しては、sudo しようがしまいが、そのユーザの権限でしか読み書きができない。

この Linux のファイルシステムは「%userprofile%\AppData\Local\lxss\rootfs」に存在する。lxss ディレクトリも rootfs ディレクトリも隠しディレクトリなので注意すること。また、各ユーザのホームディレクトリは「「%userprofile%\AppData\Local\lxss\」以下に個別に用意され、Linux では /home 以下 (root だけは /root に) マップされる。

図 3.12　rootfs に Ubuntu のファイルが存在する

[*4] なお、lxrun /update を実行すると、WSL で「sudo apt-get update」が実施される。

あとがき

「やけになった人類が何をやるか、みせてやるわ」

ここ数年の Microsoft の豹変っぷりというか転向ぷりは目を見張るほどです。米国で Linux Kernel の開発コミュニティにも潜り込んでるだけではなく、日本でも毎週のごとく Linux で一旗揚げたような人が Azure に転向しております。Build の基調講演はリアルタイムに見ていたのですが、そうくるかい、と呆然としたのを覚えています。今はまだ β で、デスクトップ版の Windows にしかない、いってみれば「玩具」ですが、実装の精度が上がり、ノウハウがたまり、そして WindowsServer に実装されたらと思うと、恐ろしいものがあります。なんせ、彼らは一方で Docker をはじめとするコンテナ技術に、「足下に Linux カーネルがあればあとは全部 梱包して持ってくる」技術にも多大な投資をしてるのですから。アップルが OSS を利用しつつ、気がついたら OSS を翻弄する相手になっていたように、Microsoft は本当に、Linux を「食べて」しまうのかもしれません。

なお、本稿、とくに一章は実は別の目的、とある商業誌の原稿として作られました。が、どうも Windows の話は好まれなかったようでお蔵入りになってしまい、今回出す予定だったほんのおまけにでも使うかなとあれこれいじくってたものだったりします。本業の方が少々厄介な事態になりばたばたしているうちに時が過ぎ、気がついたらその本来の予定の、証明書周りの残りや更新、あとちょっといじってた Kerberos 周りを書く時間が全くないことに気がつき、慌てて仕上げたのがこれ、だったりします。

... 冬には、おそらくまたまともに一冊出せるかな、と。

Windows Subsystem for Linux, Linux を食べた Windows

著　者　千種 菊理

発行所　Nagisa unworks

Nagisa unworks.