WSL

Windows Subsystem for Linux

構築と利用

Windows 10で利用するLinux環境

川口直也◉著

CUTT
カットシステム

・本書の内容についてのご意見、ご質問は、お名前、ご連絡先を明記のうえ、小社出版部宛文書（郵送またはE-mail）でお送りください。
・電話によるお問い合わせはお受けできません。
・本書の解説範囲を越える内容のご質問や、本書の内容と無関係なご質問にはお答えできません。
・匿名のフリーメールアドレスからのお問い合わせには返信しかねます。

本書で取り上げられているシステム名／製品名は、一般に開発各社の登録商標／商品名です。本書では、™ および ® マークは明記していません。本書に掲載されている団体／商品に対して、その商標権を侵害する意図は一切ありません。本書で紹介しているURLや各サイトの内容は変更される場合があります。

まえがき

「Windows 上で UNIX 用のプログラムが動作する！」そんな目標を掲げたソフトウェアがこれまでいく度も登場してきました。筆者はかつて Cygwin にお世話になった世代です。同世代の人には CoLinux を選択した方もいたでしょう。かつて、VM がまだメジャーになっていない時代に計算機のリソース消費を抑えつつ異種 OS のプログラムを実行させる取り組みがいくつもありました。

それから長い年月が経ちましたが、Windows Subsystem for Linux（WSL）はそうした取り組みの集大成のようなソフトウェアです。WSL では本家の Ubuntu が動作します。また、`apt` コマンドでリポジトリから Linux 用のプログラムを直接ダウンロードしてインストールすることが可能です。かつて、Windows 上で UNIX 用プログラムを実行できるソフトで、これほどの完成度を持ったものは存在しませんでした。

筆者は本書の執筆にあたり、WSL の魅力を広範に伝えつつ、なるべく定量的なアプローチで WSL の完成度を測るよう試みました。また、Linux でも Windows でもない、WSL 独自の設定項目などについても詳細に解説を行うことを目標にしました。

献辞

私にこの本を出版する機会を与えてくれた、石塚 勝敏氏に謝意を表します。

本書の読者層について

本書は、Windows アプリケーションの開発者や Windows 環境のシステム管理者などを読者対象としています。そのため、コンソールでのコマンド操作時のカレントディレクトリの確認、コマンド実行によりエラーが発生した際にエラーメッセージによって必要なパッケージのインストールができること、Linux コマンドの詳細を検索できることなどを、前提としています。

目次

まえがき iii

第1章　はじめに 1

1.1　Windows Subsystem for Linux（WSL）とは 2
1.1.1　WSL の歴史 2
1.1.2　WSL のアーキテクチャ 4
1.2　既存のシステムとの比較 6
1.2.1　Cygwin 6
1.2.2　MinGW 6
1.2.3　MSYS/MSYS2 7
1.2.4　ユーザーモードエミュレータ 7
1.2.5　X Windows System 8
1.3　WSL の実用性 9
1.3.1　WSL でできること 10
1.3.2　WSL でできないこと 11
1.4　本書の構成とねらい 12
1.4.1　本書の構成 12
1.4.2　本書のねらい 12
1.4.3　本書で明らかにすること 13
1.5　動作環境 14

第2章　セットアップ 17

2.1　WSL の有効化 18
2.2　ディストリビューション（Ubuntu）のインストール 19
2.2.1　複数のディストリビューションのインストール 20
2.3　コンソールの起動 21
2.4　ユーザーの追加と削除 23
2.4.1　ユーザーの追加 23
2.4.2　ユーザーの削除 24
2.5　WSL の更新 25
2.6　WSL の削除 25
2.7　ディストリビューションの修復 27
2.7.1　修復 29
2.7.2　リセット 29

2.8 ディストリビューションのインストール先 29

第3章 ファイルシステムとその構成 31

3.1 WSL のファイルシステム 32
3.1.1 VolFs 32
3.1.2 DrvFs 33
3.2 WSL のファイル構成 34
3.3 Windows から見た WSL ファイル 37
3.4 カレントディレクトリ 37
3.5 パス 38

第4章 システム管理 41

4.1 システム状態の取得 42
4.1.1 CPU 42
4.1.2 メモリ 47
4.1.3 I/O 49
4.1.4 ネットワーク 50
4.1.5 /proc ファイルシステム 56
4.1.6 ログ 58
4.1.7 その他のシステム情報 59
4.2 デバイス制御 62
4.2.1 デバイスファイル 62
4.2.2 WSL でサポートしていない操作 63
4.3 言語ロケール 64
4.4 プリンタ 65

第5章 コマンド 67

5.1 主要 UNIX コマンド 68
5.1.1 ファイルディレクトリ管理コマンド 68
5.1.2 システム状態 71
5.1.3 テキスト処理 73
5.1.4 ソフトウェア開発 76
5.1.5 セッションの終了 77
5.2 圧縮解凍 77
5.3 ネットワーク 78
5.4 エディタ 81
5.5 印刷 82

5.6 その他 84
5.7 シェル 85
5.7.1 環境変数 85
5.7.2 bash 88
5.7.3 zsh 92
5.8 Windows コマンドとの連携 93
5.8.1 wsl コマンド 93
5.8.2 wslpath コマンド 94
5.8.3 WSL から Windows のプログラムを実行 95
5.8.4 応用例 96
5.8.5 bash.exe 97
5.9 テストスイートによる検証 98
5.9.1 BusyBox のテストスイート 98
5.9.2 Cureutils のテストスイート 101

■ 第 6 章 サービス 105

6.1 Web サーバー 106
6.1.1 apache 106
6.1.2 nginx 108
6.2 SSH サーバー 109
6.2.1 openssh-server 109
6.2.2 Windows からの SSH 接続 110
6.3 FTP サーバー 111
6.3.1 ProFTPD 111
6.3.2 Pure-FTPd 112
6.4 サービスの自動起動 113

■ 第 7 章 GUI 115

7.1 X Window System とは 116
7.1.1 WSL と X Window System 116
7.2 X サーバー 117
7.3 GUI プログラム 118

■ 第 8 章 プログラミング 123

8.1 システムコール 124
8.1.1 WSL に実装されたシステムコール 125
8.1.2 POSIX システムコール 136
8.1.3 Linux Test Project（LTP）の実行結果 137

8.2 Linux ネイティブコードプログラミング 143
8.2.1 アセンブラによるシステムコール呼び出し 143
8.2.2 C 言語 145
8.3 クロスコンパイル環境の構築 156
8.3.1 Windows 用クロスコンパイラの環境構築 156
8.3.2 Raspberry Pi 用クロスコンパイラの環境構築 157
8.4 スクリプト言語 157
8.4.1 Ruby 157
8.4.2 R 言語 162
8.5 Java 164
8.5.1 OpenJDK 164
8.5.2 Java Native Access（JNA） 165
8.5.3 GraalVM 167
8.6 Windows/WSL プログラムの連携 172
8.6.1 メモリマップドファイル 172
8.6.2 wslapi 176
8.6.3 Windows Script Host（WSH）との連携 184
8.6.4 WMI 189
8.6.5 その他の Windows・WSL 連携プログラム 191

■ 第 9 章 応用アプリケーション 195

9.1 事前準備 196
9.1.1 高機能な X サーバー 196
9.1.2 日本語入力 200
9.2 GNOME 端末 202
9.3 GNOME オフィス 204
9.3.1 AbiWord 204
9.3.2 Gnumeric 205
9.3.3 Evince 205
9.4 開発環境 207
9.4.1 LLVM 207
9.4.2 COINS 208
9.4.3 TCC 209
9.5 エディタ 211
9.5.1 Emacs 211
9.5.2 gedit 212
9.5.3 Visual Studio Code 213
9.6 印刷 214
9.6.1 LaTex 214
9.6.2 Ghostscript 217
9.7 システムソフトウェア 218

9.7.1 QEMU 218
9.7.2 Wine 222
9.7.3 SPIM 223
9.7.4 memcached 224
9.8 Web ブラウザ 226
9.8.1 midori 226

■ 第 10 章 性能測定 231

10.1 コマンドの処理性能 232
10.2 システムコールの処理性能 236
10.3 演算性能 242

■ 付録 A WSL2 247

A.1 WSL2 のインストール方法 248
A.2 WSL2 でできること 249
A.3 応用アプリケーション 254

索 引 265

第1章

はじめに

本書は Windows Subsystem for Linux（WSL）の解説書です。本書では、WSL の魅力を伝えながら、その使い方を実用的なアプローチで解説することを目標に記述しています。

1.1 Windows Subsystem for Linux（WSL）とは

Windows Subsystem for Linux（WSL）は、Windows 上で Linux のユーザーモードプログラム（ELF バイナリ）を動作させることが可能なサブシステムです。例えば、Linux の `ls` コマンドや `cat` コマンドをネイティブな ELF バイナリのまま Windows 上で実行することができます。これまでにも UNIX 用のプログラムを Windows 上で動作させることを目的としたアプローチがいくつか存在しました。しかし、Linux 用の ELF バイナリを変更することなく、WSL ほどの完成度をもって実行することができるアプローチは、WSL が初めてです。

1.1.1 WSL の歴史

ここでは WSL の歴史について解説します。まずは、WSL が登場する以前の Microsoft 製品について解説し、WSL とその後継となるであろう WSL2 についても言及します。

Microsoft Windows Services for UNIX

Microsoft Windows Services for UNIX（SFU）[1] は、Windows NT から導入された、POSIX API を Windows 上で実行するための機構とそのサブシステムです。SFU のサブシステムは Windows カーネルの上位層に位置付けられており、POSIX API を Win32 API とは別の枠組みで実行することができます。これにより、Win32 API で提供していない機能を、Windows カーネルに対して直接要求することができます。

SFU は Microsoft が公式にリリースする製品であり、それなりに利便性の高いシステムでした。しかし、UNIX 用のプログラムを Windows 上で実行するには SFU が提供するコンパイラを用いて再コンパイルする必要があり、また SFU が提供するシステムコールの数が少ないこともあって、なかなか Linux 用の OSS を SFU 用に移植するには苦労を伴いました。

SFU は、Subsystem for UNIX-based Applications（SUA）[2] という製品に引き継がれ、その後 Microsoft がサポートする UNIX 環境の系譜はいったん途切れていました。

Bash on Ubuntu on Windows

Bash on Ubuntu on Windows [3] は、2016 年に発表された WSL の前身となったシステムです。Windows 上で Ubuntu 用の ELF バイナリが動作する仕組みを、Windows のサブシス

テムである Windows Subsystem for Linux（Beta）により提供します。WSL の名称の末尾に Beta が付いていることからも分かるとおり、Bash on Ubuntu on Windows というシステムは実験的なものでした。例えば、Windows 10 を開発者モードで実行しないと WSL をインストールできないといった制限が当初はありました。また、Bash on Ubuntu on Windows ではその名前のとおり、インストールできるディストリビューションが Ubuntu に限定されていました。

Bash on Ubuntu on Windows の Windows サブシステムのアーキテクチャは現在の WSL とほぼ共通です。これは、前述した SFU と同様に Windows カーネルの上に Linux 用のサブシステムを構築し、そのサブシステムが Linux 用プログラムの実行をサポートするシステムです。

Windows Subsystem for Linux（WSL）

Windows Subsystem for Linux（WSL）は Windows 10 Fall Creators Update において、Bash on Ubuntu on Windows からその名称が変更されたものです [4]。名称が変更された理由として、Ubuntu 以外に SUSE などのディストリビューションをサポートすることになったからだと Microsoft は説明しています。各ディストリビューションは独立してインストールすることができ、Windows 上に複数の Linux 環境を同時に立ち上げることができます。

現在、WSL は Windows に標準搭載されており、開発者モードに変更することなく「Windows の機能の有効化」によって有効化することができます。追加でコンポーネントをインストールする必要がないため、セキュリティ要件が厳しい企業内での使用にも向いています。

現在の WSL の制限として、サポートされているシステムコールが完全ではない点が挙げられます。つまり、Windows のサブシステムとして、Windows のカーネル上で実行できないシステムコールはサポートされていません。このため、一部のプログラム（例えばディスクに低レベルでアクセスする dd コマンド）は動作しません。

Windows Subsystem for Linux 2（WSL2）

Windows Subsystem for Linux 2（WSL2）は次世代の WSL です。本書執筆時点（2019/9）ではまだ正式リリースされていません。WSL2 は Windows 10 の Build 18917（20H1）においてリリースされる見込みです [5]。

WSL2 の特徴は、Hyper-V 上で本物の Linux カーネルを実行する点にあります。WSL では

Windows カーネル上で Linux のサブシステムを実行し、そのサブシステム上で Linux のユーザーモードプログラムを実行していました。これに対し、WSL2 では Hyper-V の上で本物の Linux カーネルを実行し、その上で Linux のユーザーモードプログラムを実行します。ユーザーモードプログラムの入出力は、Windows 上で動作するユーザーモードのプロセスと連携して処理されます。これにより、Windows 側のプログラムと Hyper-V 上で動作する Linux アプリケーションの連携を可能とします。

このような複雑なアーキテクチャを採用する理由として Microsoft は「完全なシステムコールの互換性」を挙げています [6]。

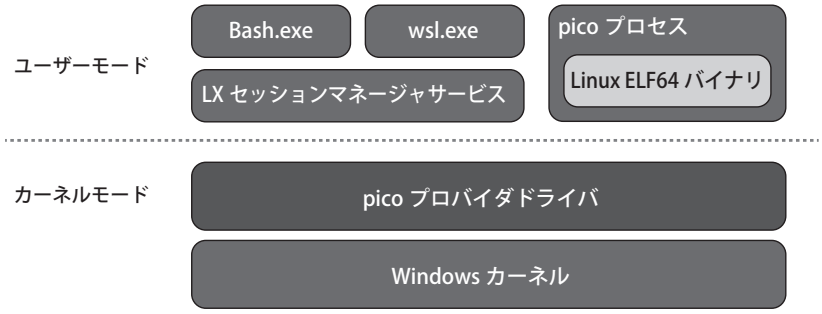
なお、本書は WSL の解説書であり WSL2 については今後の展望について言及するにとどまります。

1.1.2　WSL のアーキテクチャ

WSL のアーキテクチャの概要は、Microsoft 社が公開しているブログ（Windows Subsystem for Linux Overview [7]）によると次のとおりです（図 1.1）。

> WSL はカーネルモードドライバ（pico プロバイダドライバ）と、Windows のユーザーモードアプリケーション（`Bash.exe`, `wsl.exe`）、Windows のサービス（LX セッションマネージャサービス）、Linux プログラムを実行する pico プロセスで構成されています。

図 1.1　WSL のアーキテクチャの概要



カーネルモードドライバである、pico プロバイダドライバは、Linux のシステムコールをエ

ミュレーションするカーネルモードのレイヤーです。

`Bash.exe` と `wsl.exe` は Windows の Linux プログラムを起動するためのトリガーとして機能する Windows のユーザーモードプログラムです。`Bash.exe` と `wsl.exe` はともに指定された Linux プログラムを起動するように LX セッションマネージャサービスに通知します。

pico プロセスは Windows のユーザーモード上で実行される Windows プロセスです。pico プロセスの内部では AMD64 の機械語命令が実行され、システムコールの呼び出しが発生すると通常の Windows のシステムコール呼び出しではなく、pico プロバイダドライバに対する呼び出しとしてシステムコールを呼び出します。pico プロセスが実行する ELF バイナリは 64bit の Linux 用 ELF バイナリです。

ファイルシステム

WSL では Linux のファイルシステム操作を Windows のカーネルモード API による操作でエミュレートしています。

WSL では Linux のファイルシステム空間を再現しつつ、Windows のファイルシステム空間にアクセスするために、以下に示す VolFs と DrvFs という 2 つのファイルシステムを用いています。

VolFs は `/root` や `/home` といったディレクトリにマウントされるファイルシステムです。VolFs の特徴は完全な Linux のファイルシステムのエミュレーションが可能な点にあります。VolFs 上では Linux のパーミッションが設定でき、ハードリンクを使用することができます。また、ファイル名についても Windows の制約を受けないファイル名を使用することができます。

DrvFs は、Windows のドライブをマウントするためのファイルシステムです。DrvFs は Windows と連携することを目的に使用されるため以下の制限があります。

- Windows 互換のファイル名しか使用できない
- Linux のパーミッションが使用できない

1.2 既存のシステムとの比較

ここでは WSL と既存のシステムやソフトウェアとの違いについて解説します。

1.2.1 Cygwin

Cygwin [8] [9] は、UNIX 用のアプリケーションを Windows 上で実行するためのソフトウェアと、そのソフトウェアを利用して動作する多数の UNIX 用のプログラムがあらかじめ用意されたシステムを指します。Cygin のコアは `cygwin1.dll` という DLL ファイルで構成されており、この DLL ファイルの中には POSIX API を Windows の API でエミュレートするコードが実装されています。すべての Cygwin 用アプリケーションは `cygwin1.dll` をリンクすることで UNIX 用のプログラムを実行することができ、それらのアプリケーションを呼び出すためのインターフェースとして Cygwin 専用のコンソールが用意されています。

Cygwin 用のアプリケーションを開発するには、Cygwin 上で動作する専用の `gcc` を利用してソースコードをコンパイルする必要があります。このため、既存の Linux 用アプリケーションの ELF バイナリを直接 Cygwin 上で実行することはできません。

1.2.2 MinGW

MinGW（Minimalist GNU for Windows）[10] は、Windows 上で動作する `gcc` を含むツールチェインです。MinGW は Cygwin とは異なり、POSIX API をエミュレーションする機能を提供していません。このため、UNIX 用アプリケーションを移植する場合は `libc` レベルで Windows に適合させる必要があります。具体的には、Windows の `libc` ライブラリである `msvcrt.dll` をリンクする必要があります。したがって、UNIX 用アプリケーションを MinGW でコンパイルするには、たいていの場合ソースコードレベルでの修正が必要であったり、使用しているシステムコールによってはその機能を実現できなかったりします。

MinGW のコマンドは、WSL や Cygwin と違い Windows のコンソールから直接呼び出すことができます。このため、通常の Windows アプリケーションとして `gcc` を実行することができます。

MinGW は WSL と違い、あくまでも `gcc` を含んだツールチェインであり、UNIX のシステ

ムコールをエミュレートする仕組みはありません。

1.2.3　MSYS/MSYS2

MSYS [11] は MinGW と連携して動作するユーティリティ群です。bash や awk、gzip などのユーティリティコマンドを集めてパッケージ化しています。MSYS は MinGW がなくても単体でも動作します。このため、gcc は不要だが bash や awk といったユーティリティが必要な場合にも用いられます。

MSYS2 [12] は MSYS から独立したプロジェクトであり、Windows ネイティブアプリケーションとの連携性が向上するように MSYS のコードを再実装しています。MSYS2 は Bash を実行するために専用のコンソールが用意されています。ただし、MSYS2 のコマンドを通常の Windows コンソールから実行することもできます。MSYS2 の特筆すべき点として dd コマンドが動作することが挙げられます。WSL は低レベルの I/O を扱う dd コマンドは動作しませんが、Windows の API を直接呼び出す MSYS2 では動作します。

MSYS も MSYS2 もその利点は UNIX 用のコマンドユーティリティを Windows 上で利用できることにあります。ただし、POSIX API のエミュレーションは行わず、コンパイラも単独では提供していないため、既存の UNIX アプリケーションを移植することは単独ではできません。

1.2.4　ユーザーモードエミュレータ

ユーザーモードエミュレータは、ユーザーモードの機械語命令だけをエミュレートし、システムコールはエミュレータが動作している環境の資源を利用する形態のエミュレータです。

ユーザーモードエミュレータの代表例は、QEMU のユーザーモードエミュレーション機能 [13] です。QEMU はハードウェア全体をエミュレートするフルシステムエミュレーション機能とユーザーモードの機械語命令のみをエミュレーションするユーザーモードエミュレーション機能を提供しています。QEMU のユーザーモードエミュレーション機能では、Linux 用の ELF バイナリを他の CPU アーキテクチャ上で実行することができます。

NestedVM [14] は、MIPS R2000 用バイナリを対象とする、Java で書かれたユーザーモードエミュレータです。ここでいう MIPS R2000 用バイナリは、gcc のクロスコンパイルオプション「mips-unknown-elf」が適用され、NestedVM 独自のシステムコール番号に基づくシ

ステムコールを使用し、NestedVM が定義するエントリポイントから開始される命令列からなる、静的リンクされた ELF バイナリです。したがって、MIPS/Linux 等の汎用 OS 上で一般的に使用されるアプリケーションバイナリは直接実行できません。既存のアプリケーションを NestedVM 上で利用する場合には、この特異な環境をターゲットとして再コンパイルする必要があります。NestedVM の特徴は JavaVM 上で動作する点にあります。このため、実行する環境を選ばずにエミュレーションを行うことができます。

EspressoVM [15] は、筆者が研究のために実装した JavaVM 上で動作するユーザーモードエミュレータです。EspressoVM は性能面では他のエミュレータに比べてかなり遅いですが、機能面では共有ライブラリの動的リンクや `xlib` を用いた GUI プログラムの実行などをサポートしており、さらに x86/Linux 用の GUI アプリケーション（`xlock` など）を Java アプレットとして Web ブラウザ上で実行することができます。EspressoVM の下層ではエミュレートする対象のシステムのシステムコールの差異をある程度吸収する機構が実装されており、Linux の ELF バイナリを、Cygwin を用いて実行することができます。

ほかにも Veels [16]、EM86 [17] といったユーザーモードエミュレータが存在します。

ユーザーモードエミュレータの特徴は、ユーザーモードの機械語命令のみをエミュレートし、システムコールはエミュレータの実行環境の資源を利用する点にあります。この仕組みのメリットはエミュレートする対象のアプリケーションと、エミュレータが動作する環境のリソースの連携が取りやすい点にあります。

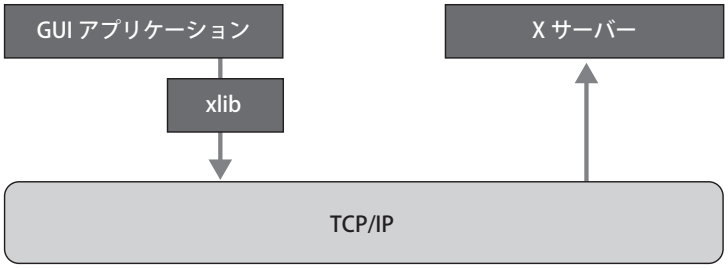
ユーザーモードエミュレータの難点は、性能面でオーバーヘッドが大きくなりがちな点と、システムコールの仕様がある程度似通ったシステムでないと実装ができない点にあります。

ユーザーモードエミュレータは、WSL が機械語命令のエミュレーションを行わずシステムコールのエミュレーションのみを行うのと逆のアプローチで、アプリケーションの可搬的実行をサポートしています。

1.2.5　X Windows System

X Window System [18] は仮想化機構ではありませんが、リモートの Linux 環境で実行している GUI アプリケーションを、あたかもローカルで実行しているデスクトップアプリケーションのように操作することができます（図 1.2）。例えば、Linux 上で動作している GUI アプリケーションを別のマシン上の Windows アプリケーションのように操作することができます。

図 1.2　X Window System



X Window Systemに基づいてXアプリケーションを描画するXサーバーというソフトには、シングルウィンドウモードとマルチウィンドウモードの２つのモードが用意されています。このうち、シングルウィンドウモードは単一のウィンドウの中に複数のXアプリケーションを描画する方式で、Windows のリモートデスクトップに近い形態で操作します。

マルチウィンドウモードは、Xアプリケーションに対して個別のウィンドウを提供する機能で、この方式を利用するとリモートのXアプリケーションがあたかもローカルで実行中のアプリケーションであるかのように操作することができます。

このX Window System を利用すれば、Linux 用の GUI アプリケーションを Windows アプリケーションのように操作することができます。しかし、システムコールは完全に別のシステムに対する要求として実行されることから、Windows システムとの連携性は WSL に比べて大きく劣ります。

1.3　WSL の実用性

WSL はほとんどの Linux 用コマンドを実行することができ、その仕組み上ファイルなどの Windows 上の資源に対して Linux プロセスからシームレスにアクセスすることができるため、Windows と Linux で高い連携性を発揮しています。

一方、低レベルの I/O を要求する dd コマンドなど WSL では利用できない機能も存在します。

ここでは WSL でできることとできないことを整理します。

1.3.1　WSL でできること

WSL では、ファイルなどの Windows 上の資源を Linux のコマンドから利用することができます。例えば、WSL 上で ls コマンドを実行すると Windows の C ドライブ上のファイルの一覧を取得することができます。

また、標準入出力を Windows 側のシェルと共有することができ、例えば Windows のコンソール（コマンドプロンプト）から以下のようなコマンドを実行することができます。

以下、「C:¥>」は Windows のコマンドプロンプト、「$」「#」「%」は WSL の bash 等のシェルです。なお、> や $、#、% はプロンプトといいます。

```
C:¥> echo Hello | wsl cat
Hello
```

上記のコマンドラインは、Windows のコンソール上で echo コマンドを実行し "Hello" という文字列を標準出力し、その結果を WSL 上の cat コマンドにパイプで渡してコンソール上に表示するものです。wsl コマンドについてはのちに解説します。このように WSL のコマンドラインは Windows と高い連携性があります。

また、WSL から Windows のアプリケーションを以下のように起動することができます。

```
$ notepad.exe
```

上記の例では、Windows のメモ帳を WSL のコンソール上から起動しています。

ファイルについては、WSL 側からはデフォルトで Windows の各ドライブがマウントされた状態となっており、各種コマンドからシームレスにアクセスすることができます。Windows 側から WSL のファイルにアクセスするには ¥¥wsl$¥Ubuntu-18.04¥ のような接頭辞のついた 9P プロトコルによるファイル共有としてアクセスする必要があります。このパスには Windows のコンソールから dir コマンドなどでアクセスすることができます。