

本書は、小社刊の以下の刊行物をもとに、大幅に加筆と修正を行い書籍化したものです。

- ・『WEB+DB PRESS』Vol.120 特集「[自作 OS × 自作ブラウザで学ぶ] Web ページが表示されるまで——HTML を運ぶプロトコルとシステムコールの裏側」

---

本書の内容に基づく運用結果について、著者、ソフトウェアの開発元および提供元、株式会社技術評論社は一切の責任を負いかねますので、あらかじめご了承ください。

本書に記載されている情報は、特に断りがない限り、執筆時点（2025 年）の情報に基づいています。ご使用時には変更されている可能性がありますのでご注意ください。

本書に記載されている会社名・製品名は、一般に各社の登録商標または商標です。本書中では、™、©、® マークなどは表示していません。

# はじめに

アプリケーションの動作を陰ながら支える縁の下の力持ち、それがオペレーティングシステム (OS) です。

みなさんの身近にあるコンピューターのほとんどは、OS なしでは単なる電子回路の塊になってしまいます。それにもかかわらず、OS がどのようなことをしているのか、なぜ OS が必要なのか、その正体はあまり知られていないのが現状です。

本書は、最低限の機能を持った OS を手作りすることを通して、みなさんに OS の果たす役割とそのしくみについて理解していただくことを目標としています。最終的には、別冊の『[作って学ぶ] ブラウザのしくみ』<sup>注1</sup> で開発している単純な自作ブラウザが動作するようにします。

OS が果たすべき 2 つの大きな役割は、資源の管理と、ハードウェアの抽象化です。これら 2 つの題材を柱として、本書ではメモリ管理機構やプロセス管理、文字の入出力や GUI、さらには USB デバイスのサポートやネットワーク通信プロトコルまで、実際に動作可能なコードを示しつつ解説します。これにより机上の理論だけでなく、実際に手を動かして動くものを作り上げながら、そのしくみに対する理解を深めていきます。そして最終的には、インターネット上の Web サイトから HTML を取得できるところまで機能を発展させ、アプリケーションがどのように動作してネットワーク通信を行うのかを、上から下まで体験します。

本書で実装する OS は、一般に広く用いられている x86\_64 アーキテクチャを動作対象としています。またメモリ安全性が高く、それでいて低レイヤなプログラミングを行いやすいプログラミング言語 Rust を実装言語に採用しました。開発環境については Linux を例に解説しますが、それ以外の主要な OS でも動作確認ができるよう、必要に応じて注釈を入れたり、サポートページでの支援を提供したりします。また、基本的にはエミュレーターを使用して動作確認をします

---

注1 土井麻未著『[作って学ぶ] ブラウザのしくみ——HTTP、HTML、CSS、JavaScript の裏側』技術評論社、2024 年

ので、1 台の PC のみで読み進めていただけるようになっています。

決して簡単とは言い切れない内容ではありますが、普段は Web アプリケーションのように高レイヤの開発をされている方々や、まだプログラミングを始めたばかりという学生の方々でも理解できるよう、できる限りわかりやすく解説するよう心がけています。そのため、少しでも OS というものに興味を持っている方であれば、本書を楽しんでいただけるはずです。

本書をきっかけに、OS という分野のおもしろさに一人でも多くの方が気付いてくだされば、筆者はたいへんうれしく思います。

ようこそ OS の世界へ！ ぜひ楽しんでってください。

**hikalium**

# 本書の読み方

## サンプルコードのダウンロード

本書で紹介するコードは紙面の都合上、実際のソースコードの一部になっている場合があります。完全なソースコードについては GitHub の hikalium/wasabi<sup>注1</sup> より入手できます。デフォルトで表示される main ブランチは今後の開発で変更される可能性があるため、本書のために用意したブランチ wasabi4b を参照してください。以下のコマンドを実行すると wasabi4b ブランチを wasabi というディレクトリ名で、カレントディレクトリに clone できます。

```
$ git clone -b wasabi4b https://github.com/hikalium/wasabi.git
```

Git に慣れていない方は直接上記ブランチをチェックアウトしていただいて OK です。もし不慣れな場合は、ZIP ファイルを GitHub よりダウンロードできますので、適宜ご活用ください。

## 想定読者

本書は、できる限り多くの方にわかりやすいよう気を付けて書いていますが、以下のような方々を想定して執筆しました。

- ・プログラミングは少しかじったことがある
- ・少なくとも 1 つのプログラミング言語をある程度使える
- ・ Rust というプログラミング言語の名前は聞いたことがあるが書いたことはない
- ・ OS という言葉は耳にしたことがある
- ・しかし OS とは何かと問われると詰まってしまう
- ・ OS とアプリケーションの差が何なのかよくわからない
- ・だけど OS とは何か知りたい

注 1 <https://github.com/hikalium/wasabi>

## ■本書の読み方

- ・ハードウェアとソフトウェアが別々のものだという事はわかる

もちろん、じっくりと時間をかけて読めば、上記に当てはまらない方でも楽しめるはずです。

---

## 本書の構成

---

本書は、I、II巻の分冊で構成されているうちのI巻にあたります。II巻については、2025年の刊行を予定しています。

I巻（本書）で解説する内容は、以下のとおりです。

- ・OSとは何か
- ・ベアメタルプログラミングの方法
- ・メモリ管理の実装
- ・マルチタスクと例外処理の実装
- ・ハードウェアの制御

ここまでの内容をすべて実装すれば、USBキーボードからのコマンド入力を受け付け、コマンドに応じて画面に図形や文字を描画できるようになります。またメモリ管理やマルチタスクのような基本的な計算資源の分配も行われるようになり、コンピューター黎明期のOSのようなものが出来上がります。

II巻ではI巻の内容をさらに発展させ、以下の内容について解説する予定です。

- ・GUI
- ・アプリケーションの実行とシステムコール
- ・ネットワークとプロトコル

I、II巻すべての内容を実装し終われば、キーボード入力を主とするCUIだけでなく、画面上に描画された図形とマウスを組み合わせた視覚的・直感的な入力方法であるGUIの実装や、OSとは独立したプログラムであるアプリケーションの実行と、それに対してOSが機能を提供するためのしくみであるシステムコール、そして最終的にはネットワーク通信と各種インターネットプロトコル

を実装することにより、単一のコンピューターの枠を超えて遠く離れた別のコンピューターと通信するところまで、現代のコンピューター・システムを深く広く理解できる内容となっています。

---

## 関連書籍について

---

本書の内容は、2024年発売の『[作って学ぶ] ブラウザのしくみ』と深く連動しています。

『ブラウザのしくみ』では、GUIを備えたシンプルなブラウザを実装することで、そのしくみを理解します。また、ここで実装するブラウザは、本書『OSのしくみ』の上で動作するアプリケーションとして開発されているため、『ブラウザのしくみ』を読破した方にとっては、本書を読むことで、さらに下側で動くOSのしくみまで理解を深めることができます。また、本書『OSのしくみ』を先に読まれる方にとっては、OSがハードウェアを抽象化して提供する機能の数々が、ブラウザという身近なアプリケーションでどのように活用されるのか、さらに高い視点から一望できる機会となるはずです。

どちらの書籍も、単体で十二分に読みごたえのある内容となっていますが、これらの書籍を並べて読むことでコンピューターというものに対する認識の解像度が飛躍的に高まること請け合いですから、焦らずゆっくりと読み進めていただければ幸いです。

---

## 開発環境のセットアップ

---

本書で解説している手順は、ChromeOS上のDebian環境で動作することを確認しています。

```
$ cat /etc/debian_version  
12.7
```

したがって、同じくDebian系のOSであるUbuntuなどでも、ほぼ同様の手順で動作させることができます。

それ以外の環境で本書の内容を試す場合の詳しい手順については、サポートページを随時更新しますので、そちらをご確認ください。

<https://lowlayergirls.github.io/wasabi-help/>

## ● Rust ツールチェーンのインストール

Rust コンパイラや関連ツールは、`rustup` を利用してインストールできます。  
<https://rustup.rs/> にアクセスして、そこに書かれているコマンドを実行することで、必要なツールをインストールできます。

まずはターミナルを開いて、以下のように入力して、Enter キーを押します。

```
$ curl --proto 'https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

いくつかカスタマイズのための質問が出てきますが、すべてデフォルトで問題ないので、Enter キーを連打してください。

インストールが終わったら、一度シェルを再起動するか、ターミナルウィンドウを開き直してください。この手順を飛ばすと環境変数が反映されず、Rust ツールチェーンを呼び出すことができません。

シェルを開き直したら、`cargo --version` および `rustc --version` を入力して、これらのコマンドがインストールされていることを確かめてください。

```
$ cargo --version
cargo 1.82.0 (8f40fc59f 2024-08-21)

$ rustc --version
rustc 1.82.0 (f6e511eec 2024-10-15)
```

バージョン番号については、現段階では一致しなくても問題ありません。適切なバージョンを指定する方法については第 2 章で解説します。

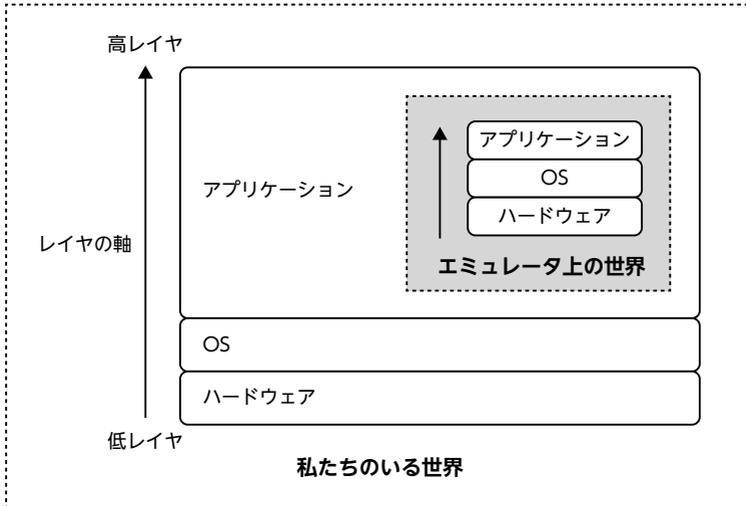
## ● PC エミュレーター QEMU のインストール

本書では、QEMU というエミュレーターを利用して自作 OS を動作させます。QEMU は、コンピューターの上でコンピューターを仮想的に動かすためのアプリケーションです。

本来の OS は、ハードウェアを制御するという役割を果たすために、ハードウェアの上で直接動作させることがほとんどです。しかし、現代のハードウェアは多種多様であり、そのすべてで動作する OS の作り方を解説することは、入門の範囲を大きく超えてしまいます。そこで、ある実在するコンピューターの動作を模

做するようなプログラムの上で OS を動作させることにより、どの読者の方でも、実質的に同じ「コンピューター」を対象とするような OS の作り方を体感できます (図 0-1)。

図 0-1 エミュレーターの世界は外の世界を模倣している



ええっ！ そんなの虚構じゃないか！ と思われる方もいらっしゃるかもしれませんが、安心してください。実際に、エミュレーター上で OS を開発するという手法は、現実世界で使われている OS の開発現場でも用いられる一般的な方法です。また、対応するハードウェアをお持ちであれば、本書で実装した OS をハードウェアの上で直接起動してみることも可能です。具体的な手順については、巻末の付録で解説していますので、そちらもご確認ください。なお、ある程度実装が進むまでは、実際のハードウェアで起動しない可能性が高く、また起動したとしても、見た目では動作しているかどうかわからない可能性が高いため、まずはエミュレーターを使用した方法で開発を一度最後まで進めてみることを強くお勧めします。

Debian や Ubuntu などの apt パッケージマネージャが利用可能な環境では、以下のコマンドを実行すると、QEMU をインストールできます。

```
$ sudo apt install qemu-system-x86
```

## ■本書の読み方

インストールが完了したら、正しくインストールできているかどうか、下記のコマンドを実行して確かめてみてください。

```
$ qemu-system-x86_64 --version
QEMU emulator version 9.0.50 (v9.0.0-2247-ge2f346aa98)
Copyright (c) 2003-2024 Fabrice Bellard and the QEMU Project developers
```

例によって、バージョンは完全に一致していなくてもかまいませんが、本書執筆時点では 9.2.0 が公式で配布されている最新バージョンとなっていますので、メジャーバージョンが 9 以上であれば問題なく動作するはずです。

---

## 本書のコードの読み方

---

本書で実装していく OS のコードは以下のように書かれています。

```
src/main.rs
fn main() {
    println!("Hello, world!");
}

#![no_std]
#![no_main]

#![no_mangle]
fn efi_main() {
    //println!("Hello, world!");
    loop {}
}

use core::panic::PanicInfo;

#[panic_handler]
fn panic(_info: &PanicInfo) -> ! {
    loop {}
}
```

それぞれの意味は以下のとおりです。

- 太字  
新規に追記する部分
- 取り消し線  
削除する部分

- ・それ以外  
前に実装した部分

---

## 参照している仕様書の表記方法

---

参照している仕様書は以下のように仕様書を表す記号を記入しています。

これは ACPI 仕様書の「21.1 Types of ACPI Data Tables」<sup>[acpi\_6.5a]</sup>を読むとわかるのですが、MCFG の定義は ACPI 仕様の範囲外なのです。

記号がどの仕様書を表すかは、巻末の「参照している仕様書の一覧」を参照してください。

## 目次 CONTENTS

はじめに.....	iii
本書の読み方.....	v
サンプルコードのダウンロード.....	v
想定読者.....	v
本書の構成.....	vi
関連書籍について.....	vii
開発環境のセットアップ.....	vii
本書のコードの読み方.....	x
参照している仕様書の表記方法.....	xi
目次.....	xii

## 第1章

### OS とは

——コンピューターの裏側を支えるソフトウェアを知る.....	1
OS とは何か.....	2
日常にある OS の例.....	2
OS とコンピューターの関係.....	3
コンピューターの基本的なしくみ.....	3
アプリケーション——人々がコンピューターを使う理由.....	6
OS ——ハードウェアとアプリケーションの狭間で.....	7
本書で実装する OS の全体像.....	10
ベアメタルプログラミング.....	10
資源（メモリと CPU）の管理.....	11
ハードウェアの制御.....	12
本書のゴールと関連書籍の紹介.....	12
本題に入る前に.....	13

## 第 2 章

### ベアメタルプログラミングをしてみる

—— OS のない世界でプログラムを動かすための準備	15
コンピューターの構成要素	16
メモリ	17
CPU	17
入出力	18
すべてはバイナリ	19
すべてのデータは 2 進法で表現できる	19
数値をバイナリで表現する	20
16 進法は便利	23
ひとくちサイズのバイナリ、byte	24
文字列のバイナリ表現	25
画像のバイナリ表現	26
プログラムもバイナリ	29
コンパイラ——ソースコードを翻訳してバイナリを作るプログラム	33
UEFI アプリケーションを作ってみる	34
開発環境の構築	34
Hello, world を書いてみる	36
Rust ツールチェインのバージョンを固定する	38
アプリケーションと OS の違い	39
UEFI —— OS よりも前に起動する、OS を起動するためのプログラム	40
[column] 色々なファームウェア—— Legacy BIOS と UEFI BIOS	41
ターゲット——どの実行環境向けにバイナリを生成するのかコンパイラに伝える	42
QEMU を利用して UEFI アプリケーションを実行する	45
UEFI からの脱却	48
"Hello, world" はどこへ行く？	48
no_std で生きていく—— core クレートと歩むベアメタル生活	53
[column] 普段は当たり前だと思っているが実は OS が提供しているもの	57
フレームバッファに何か描く	59
Rust の便利機能を活用する	68

ビルドや実行を簡単にする.....	68
cargo clippy と HLT 命令 — CPU を無駄に回さないようにする.....	70
cargo fmt — コードをきれいに整形する.....	72
<b>もっと色々なものを描く.....</b>	<b>72</b>
四角形を描く.....	73
線分を描く.....	80
画面に文字を表示する.....	84
文字の列、文字列を表示する.....	92
<b>writeln!() マクロを使ってみる.....</b>	<b>93</b>
メモリマップを表示する.....	97
凶形描画のコードを整理する.....	107
UEFI のない世界へ行く — ExitBootServices.....	109

## 第 3 章

### メモリ管理を実装しよう

— 限りある資源を効率良く使えるようにする.....	113
<b>OS とメモリの関係.....</b>	<b>114</b>
メモリとは何か.....	114
メモリ管理とは何か.....	115
<b>実装前の準備.....</b>	<b>116</b>
ソースコードの整理 — ファイルを分割する.....	116
cargo test が通らない理由.....	140
カスタムテストフレームワークを有効にする.....	142
<b>バイト単位のアロケータを実装する.....</b>	<b>146</b>
アライメントはなぜ必要か.....	147
メモリの速度とバス幅.....	147
キャッシュ — よく使うものは近くに置こう.....	148
アライメントが合っていないと回路がづらい.....	151
簡単なメモリアロケータの実装.....	152
<b>OS のテストを Rust で書く.....</b>	<b>161</b>

シリアルポート出力の実装.....	162
[column] CONVENTIONAL_MEMORY 以外の領域の正体.....	174
デバッグを便利にする関数たちを実装する.....	176
<b>ページング——より高度なメモリ管理を行う.....</b>	<b>182</b>
ページングとは.....	182
x86_64 におけるページング.....	183
現在のページテーブルを表示してみる.....	184
動作確認のために割り込み処理・例外処理を実装する.....	194
GDT —— コンピューター黎明期、8086 時代の遺物.....	196
TSS —— 割り込み時のスタック切り替えを制御する.....	198
コードセグメントとデータセグメントの設定.....	200
割り込み関連の初期化.....	202
ブレークポイント例外のあとに実行を継続する.....	222
ページテーブルを作って設定する.....	223
ページングの動作確認をする.....	227
Pin の落とし穴.....	234

## 第 4 章

### マルチタスクを実装しよう

—— 1 つの CPU で複数の作業を並行して行う方法について知る.....	239
<b>マルチタスクとは何か.....</b>	<b>240</b>
マルチタスクの例.....	240
[column] 並行と並列の違い.....	241
簡単にマルチタスクもどきを実装してみる.....	242
<b>Rust の async/await で協調的マルチタスクをする.....</b>	<b>245</b>
async/await を使えるようにする.....	245
Future trait.....	246
Waker と RawWaker.....	247
block_on の実装.....	248
Executor.....	251
ほかのタスクに処理を譲る (yield する).....	254
時間経過を計る.....	257

タイマー——時間を計るデバイス .....	257
ACPI から HPET の場所を教えてください .....	258
HPET を初期化する .....	267
static mut を使って HPET を共有する .....	271
スレッド間で安全にデータを共有する .....	272
データ競合とは .....	272
Rust における参照のルール .....	273
Mutex ——実行時にメモリ競合を回避するしくみ .....	274
Mutex を使って HPET のインスタンスを OS 全体で共有する .....	282
タスクの実行を一定時間止める Future を作る .....	284
協調的マルチタスクの問題点 .....	287
(発展) 非協調的マルチタスク .....	287
<b>ソースコードの整理</b> .....	288
HPET の初期化処理をリファクタリングする .....	288
メモリアロケータの初期化を関数に切り出す .....	290
ページング関連のコードを整理する .....	291
画面描画周りの初期化を別の関数に切り出す .....	293
VramTextWriter を BitmapTextWriter に一般化する .....	294
print 系マクロの出力を QEMU の画面上にも表示する .....	297

## 第 5 章

### ハードウェアを制御する (1)

——デバイスを動かす方法を知る .....	301
OS とハードウェアの関係 .....	302
Port Mapped I/O と Memory Mapped I/O .....	303
Port Mapped I/O の例——シリアル入力を実装する .....	304
PCI とは .....	308
PCI の概要 .....	309
Bus、Device、Function .....	309
ベンダー ID、デバイス ID .....	310
PCI デバイスの一覧を取得する .....	311

PCI Configuration 空間 .....	311
ECAM — Enhanced Configuration Access Method .....	311
PCI デバイスの一覧を表示するコードを実装する .....	317
<b>USB コントローラ (xHCI) のドライバを実装する .....</b>	<b>324</b>
USB とは .....	324
xHCI とは .....	324
xHC の検出 .....	325
ちょっと脱線——諸々の改良 .....	325
起動時のページテーブル初期化の高速化 .....	331
Memory mapped I/O で xHC とやりとりをする .....	335
xHC のレジスタ .....	343
xHC の初期化 .....	352
xHC をリセットする .....	353
Scratchpad Buffer を確保して設定する .....	353
DCBAA を確保して設定する .....	354
Primary Event Ring を用意する .....	360
Command Ring を用意する .....	364
xHC をスタートする .....	365
IoBox —— CPU のキャッシュと Memory-mapped I/O の関係 .....	366
<b>USB デバイス接続時の処理を実装する .....</b>	<b>379</b>
イベントのポーリングをする .....	379
USB デバイスの検出 .....	385
デバイスの検出とポートの初期化 .....	391
Device Slot の有効化 .....	395
USB ポートの初期化処理を整理する .....	397
Address Device コマンド—— USB デバイスにアドレスを割り当てる .....	405

## 第 6 章

### ハードウェアを制御する (2)

—— USB デバイスを使えるようにする .....	415
<b>USB デバイスの情報を取得する .....</b>	<b>416</b>
Device Descriptor の取得 .....	416
[column] From トレイトと Into トレイト .....	428

デバイスクラス.....	428
USB における Config、Interface、Endpoint の関係.....	429
Config Descriptor とその仲間たちを取得する.....	429
<b>USB キーボードを使えるようにする.....</b>	<b>438</b>
USB キーボードの基本.....	438
[column] N キーロールオーバー.....	444
キーの押下状態から変化したキーを特定する.....	445
なぜ HashSet ではなく BTreeSet を使うのか.....	447
キーコードから文字への変換.....	448
[column] キーボードレイアウトの闇——打ちたい記号が入力できない！.....	450
<b>USB マウス……もといタブレット入力を使えるようにする.....</b>	<b>452</b>
HID レポートディスクリプタを解析する.....	478
USB タブレットの状態変化を表示する.....	502
ビットを切り出す関数を実装する.....	503
マウスボタンの状態を解釈する.....	504
ポインタ位置の情報を取り出して表示する.....	508

## ● Appendix

<b>実ハードウェアでの起動を試す.....</b>	<b>516</b>
USB メモリを FAT ファイルシステムでフォーマットする.....	516
WasabiOS を USB メモリに書き込む.....	517
USB メモリからの起動.....	519
実機で試すときの注意点.....	522
あとがき.....	523
参照している仕様書の一覧.....	526
索引.....	528
著者プロフィール.....	534

# 第 1 章

---

## OS とは

コンピューターの裏側を  
支えるソフトウェアを知る

## OS とは何か

OS (Operating System) という単語を耳にしたとき、みなさんは最初に何を思い浮かべるでしょうか？ 明確なイメージを持てる方もそうでない方も、まずは OS とは何なのか、コンピューターの中で OS はどのような役割を果たしているのか、一度おさらいしておくことにしましょう。

### 日常にある OS の例

私たちの身の回りには、コンピューターがたくさんあります。もしかすると、みなさんがコンピューターだと気付いていないものもたくさんあるかもしれません。というのも、いわゆる PC (パーソナルコンピューターの略) 以外にも、広義のコンピューター、日本語で言えば電子計算機が、現代の社会には溢<sup>あふ</sup>れているからです。スマートフォンやタブレットはわかりやすい例ですが、洗濯機や電子辞書、果てはスマートスピーカーやスマートテレビなど、多くの電化製品もコンピューターを内蔵するようになってきました。

OS は、日本語で「基本ソフトウェア」と訳されるとおり、これらコンピューターの動作に必要不可欠と言ってもよいソフトウェアです。OS とはいったい何なのかはのちほどより詳しく解説していきますので、まずは具体的な例を見ていきましょう。さまざまな企業や人々が多種多様な OS を作っています。みなさんはいくつ思い浮かびますか？

おそらく最初に頭に浮かぶのは、デスクトップ PC やノート PC でよく用いられている、Windows や macOS でしょう。また、最近は教育現場などで、ChromeOS や iPadOS が使われているのを目にすることもありません。もっと身近な例で言えば、みなさんが持っているスマートフォンのほとんどには、iOS か Android が入っていると思います。ほかにも、サーバーサイドでは Linux をベースにした OS が広く用いられています (実は ChromeOS や Android も Linux ベースの OS です)。さらに、Fuchsia という比較的最近に開発が始まった OS が、スマートスピーカーに利用されているという話を耳にした方もいらっしゃるかもしれません。

とにかくここで言いたいのは、さまざまな「OS」が世の中には存在している

ということです。

ここで一つ、みなさんに考えていただきたいことがあります。これらの OS に共通している点とは、いったい何でしょうか？

OS が動作するハードウェアもスマートフォンからサーバーまで多種多様なので、コンピューターの上で動くという以外に共通点はなさそうです。開発に使われているプログラミング言語についても、C や C++、Rust、Objective-C、Swift、Java など 1 つの OS でさえもそれぞれの部分がさまざまな言語で書かれているので、すべての OS に共通の点があるわけではなさそうです。開発している主体は、大企業だけではなく個人の開発者が集まってコミュニティベースで開発しているもの (Linux など) もあるので、一概には言えそうにありません。「ウィンドウが出てくる」とか「いろいろなアプリケーションを動かせる」という見た目や機能についても、スマートフォンでは基本的にウィンドウは表示されずし、洗濯機にアプリケーションがダウンロードできることを期待している人は多くないでしょう。

実は OS を特徴付ける性質は、私たちが普段目にする機能にあるのではなく、その裏側で静かに動作しているもっと抽象的なものにあります。キーワードは「ハードウェアの制御と抽象化」と「資源の分配」です。次のセクションから、これらについてもう少し詳しく見ていきましょう。

## OS とコンピューターの関係

コンピューターは、私たちの生活を便利にしてくれます。しかし、物理的な存在 (ハードウェア) としてのコンピューターは、あくまでも電子回路の塊であり、コンピューターを構成する要素の半分でしかありません。私たちがコンピューターから受けている恩恵の残り半分は、ソフトウェアという存在によってもたらされています。OS はソフトウェアの一種ですが、そのほかのソフトウェアやハードウェアとどのように関わっているのでしょうか？ もう少し深掘りしてみましょう。

### ● コンピューターの基本的なしくみ

一般的なコンピューターのハードウェアは、大まかに以下の 3 つの要素から構成されています。

- CPU  
計算やその制御を行う部分
- メモリ  
計算や制御に必要なデータやその結果を記憶する部分
- 入出力  
計算や制御に必要なデータやその結果を、コンピューターの外側とやりとりする部分

コンピューターは、すべての物事を数値として扱います。人間から見たら通常は数値でないもの、たとえば文字や画像なども、数値との変換規則を決めることにより数値として扱うのです。

たとえばコンピューターで日時を表現する方法の一つに、世界標準時における 1970 年 1 月 1 日午前 0 時 0 分 0 秒からの経過秒数を利用する方法があります。Linux では、`date` コマンドを利用すると、現在<sup>注 1</sup>のタイムスタンプを読み出すことができます。

```
$ date '+%s'  
1739448603
```

これを日時に変換すると、こうなります。

```
$ date --date=@1739448603'  
Thu Feb 13 09:10:03 PM JST 2025
```

次章ではもう少しほかの例についても紹介しますが、いずれにせよ、適切な変換を定義することで身の回りの情報を数値で表現できると感じていただけたのではないのでしょうか。このように、世の中の情報を数値に変換することで、数値に対してある一定の計算を行う機能しか持たない CPU があらゆる物事を取り扱うことが見かけ上はできるのです。

とはいえ、CPU 自身は、今計算しているデータが人間にとってどのような意味を持つものであるかを理解しているわけではありません。それでも CPU が意味のある計算をしているように見えるのは、CPU にどのような計算をさせるのか指示する「プログラム」が、人間によって与えられているからです。このプログラムもまた、CPU から見れば一種のデータでしかありません。

---

注 1 ここで出している例は、筆者がまさにこの文章を書いていたときの日時です。

CPU は基本的に、決まり切った計算を行うような回路しか持ち合わせていません。この計算というのはたとえば、2つの数値に対しての足し算やかけ算などの算術演算や、ビットごとの論理演算のようなシンプルで機械的なものです。それでもコンピューターが複雑な計算を行うことができるように見えるのは、これらの演算結果をプログラムの指示に従ってつなぎ合わせる作業を非常に高速に行っているからなのです。

演算結果をつなぎ合わせ次の演算へ利用するためには、どこかに結果を書いておきたくなるものです。CPU 内部には、演算の入出力に使用するための非常に高速な記憶素子である「レジスタ」というものが存在します。しかし、高速なレジスタはコストが高く、あまりたくさん用意できるものではありませんでした<sup>注2</sup>。

そこで、データを保存できる場所が大量に存在する「メモリ」という素子がCPUの外部に置かれ、ほとんどのデータはそこに置かれるようになっています。メモリは、一般的にバイト単位で場所を示す数値「アドレス」が付与されており、データを読み書きする際は、そのデータのメモリ上の場所であるアドレスも同時にメモリに伝えることで、どのデータを読み書きするのかが特定されます。

プログラムには基本的に、メモリとCPUの間でどのようにデータを移動させるか、もしくは、CPUに移動されたデータに対してどのような計算を行うか、その命令が書き連ねられています。この命令を並んでいる順番に次々と実行していくことが、CPUの基本的なお仕事です。先ほど触れたとおり、プログラムはデータの一種ですから、これらの命令列はメモリ上に置かれています。そして、次に実行すべき命令が置かれたアドレスを保持するCPU内部のレジスタが、プログラムカウンタと呼ばれるものになります。プログラムカウンタは通常、命令を実行するたびに、さらに次の命令を指すように加算されていきます。ただし、いくつかの特殊な命令は、これに当てはまりません。たとえば分岐命令は、ある特定の値にプログラムカウンタをセットします。これを用いることで、プログラムのある部分を繰り返し実行したり、スキップしたりできます。また条件分岐命令というものもあり、これは直前のレジスタの値や演算結果が、ある特定の条件を満たしているときに限り、分岐命令として働きます。

---

注2 ここていう「コスト」は、価格面はもちろんのこと、電子回路としての面積や複雑さ、消費電力、CPUの命令エンコーディング中でレジスタに言及するために必要となるビット数など、さまざまな側面が含まれています。

これらの命令を駆使することで、コンピューターは人間からの指示であるプログラムに従って、与えられる入力データに応じて複雑な処理を行うことができるのです。

さて、その処理の結果は、人間に届かなければ意味はありません。どんなにすごい計算をしていたとしても、それがコンピューターの外側に何の影響ももたらさないのであれば、コンピューターはただの高価で複雑なヒーターでしかありません<sup>注3</sup>。同様に、外界からの情報を何らかの方法でコンピューターの内側に伝えることができなければ、コンピューターは現実世界の情報を処理することはできません。つまり、コンピューターは計算をするだけでなく、その情報を外界と何とかしてやりとりする必要があるのです。

そういう意味で入出力（Input/Output、I/O）は、計算結果をコンピューターの外側に共有し、人間やほかのコンピューターとやりとりをする重要な役割を果たしています。たとえば、人間に対しての入出力の例としては、キーボード、マウス、タッチパネル、マイク、スピーカー、ディスプレイなどがあります。また、ほかのコンピューターに対しての入出力は、ネットワークインタフェースカード（NIC）や、外部記憶装置などが挙げられます。

これらの装置は、コンピューターから送られたデータに応じて、人間やほかのコンピューターが解釈可能な物理的な現象を引き起こすことで、情報を伝達します。入出力があることで、コンピューターが頑張って計算した結果を、人間やほかのコンピューターが役に立てることができるのです。

## ● アプリケーション——人々がコンピューターを使う理由

アプリケーションという単語はソフトウェアと意味が似ていると思われるかもしれませんが、実は微妙に異なります。ソフトウェアはハードウェアと対になる概念であり、人間の尺度から見たときに物理的な形を持たないものはだいたいすべてソフトウェアとすることができます。一方アプリケーションは、ある特定の目的のために作られたソフトウェア、と考えていただけるとよいと思います。

ほとんどの人は、何らかの目的を達成するための道具としてコンピューターを利用します。しかし、そのコンピューターの物理的な性質や特徴が、ユーザーの

---

注3 もちろん、寒い冬には、コンピューターの発する熱で人間が暖まることができるので、役に立っていると言えるかもしれませんね！……おや、ではコンピューターの発する熱は入出力なのでしょう？ 興味がある方は「サイドチャネルアタック」という単語で検索してみると、おもしろいかもしれません。

使用目的の達成に直接役立つことはまれです。もちろん、軽くて持ち運びやすく、性能が良く外観も好みに合うハードウェアであることに越したことはありませんが、コンピューターはハサミやトンカチとは違って物理的な作用によって役立つ道具ではなく、その上で動作するソフトウェア、つまりアプリケーションが重要なのです。

Webサイトの閲覧、表計算やプレゼンテーションの作成、動画の編集やSNSへの投稿など、ユーザーの目的にあわせて多種多様なアプリケーションを動作させることができるのが、コンピューターの強みでありユーザーの期待する価値なのです。

## ● OS——ハードウェアとアプリケーションの狭間で

ここまで、コンピューターは、物理的なハードウェアの上で、ユーザーのやりたいことをかなえるためにアプリケーションを動かす、そんな機械であると説明しました。しかし、現代のコンピューターには、もう一つ重要なソフトウェアのレイヤが存在します。それが本書のテーマである、OSです。

OSはアプリケーションと異なりユーザーの目に見えませんし、OSそれ自体がユーザーの目的の達成に直接役立つことはそれほど多くありません。もちろん、近年はユーザーの目に見える便利な機能を広義の「OS」が提供することもあります。その機能のほとんどは厳密にはOSに同梱どうこんのアプリケーションによって実現されるもので、本書で解説する狭義のOSとは少し異なります。それでも現代のコンピューターにOSがほぼ必須のものとなっているのは、コンピューターの世界が発展するに伴って、ハードウェアとアプリケーションの間の橋渡しをするOSという役割の重要性がどんどんと増してきたからなのです。

黎明期れいめいのコンピューターでは、ハードウェアの上で動くソフトウェアの目的は非常にはっきりしていました。たとえば、ENIACは最初期のプログラム可能な電子計算機ですが、これは戦時中に弾道計算を行うためにハードウェアが建造けんぞう<sup>注4</sup>され、またプログラムが変更可能であったことから、軍事的に必要とされたさまざまな計算のために利用されました<sup>注5</sup>。初期のコンピューターは、プログ

---

注4 通常ハードウェアは「製造」されるものですが、ここで「建造」と書いたのは、ENIACは部屋一つを占めるほど巨大な機械だったからです。

注5 ENIACについてもっと知りたい人は『コンピューター誕生の歴史に隠れた6人の女性プログラマー——彼女たちは当時なにを思い、どんな未来を想像したのか』（キャシー・クレイマン著、羽田昭裕訳、共立出版、2024年）を読むとさらにおもしろいかもしれません。

ラムが変更可能であったとしても、ハードウェアとソフトウェアの分離がまだそこまで進んでおらず、その両者の入り混じったコンピューターという物体そのものに明確な使用目的があった、と言えるでしょう。

その後、ある程度コンピューターが発展してくると、ハードウェアの種類もその上で計算したい問題も、どんどんと増えてきました。すると、ある問題を解くために必要となるプログラムを書く際に、過去のプログラムから再利用できる部分があることがだんだんとはっきりしてきます。たとえば、データの入出力のためのデバイス(当時はパンチカードや磁気テープドライブ)を制御するためのコードは、使い回せる部分が多くあったり、よく使われる数学的計算のコードなども使い回せたりすることが多くあります。こういった共通部分をまとめたものはライブラリと呼ばれ、同様のものは現代にも存在しています。これは言い換えれば、ハードウェアや処理の抽象化が行われはじめた、と言うことができるでしょう。

さらにコンピューターが発展すると、演算速度が向上したことで、ある問題を解くプログラムが比較的短い時間で終了するようになってきました。また同時に、コンピューターで解きたい問題も、それを解くプログラムを実行したい人も、どんどんと増えてくるようになります。しかし、コンピューターはいまだ高価なもので、ある組織に1台あるかないかというレベルでした。そうすると、一つの計算機でたくさんのプログラムを実行したくなってきます。これは、あるプログラムが終わりしだい、次のプログラムを実行していけば実現できますが、当時はプログラムをコンピューターで実行させるためには、人間が手動でパンチカードの束や磁気テープのリールを交換したり配線を組み替えたりと手作業が発生していました。これを効率化するために、人間は賢い手を考えました。そう、実行するプログラムを入れ替える作業を、プログラムにやらせたのです。まさに「面倒なことはコンピューターにやらせよう」ですね！ 具体例としては、IBM System/360のJCL(*Job Control Language*)などが挙げられます。JCLは複数人で共有して利用するコンピューターであるメインフレームにおける、プログラムの実行順や優先順位などのジョブ制御を司るコマンド群として生み出されました。これは言い換えれば、計算資源を複数のプログラムの間で分配するしくみが求められるようになってきた、ということです。この流れを汲んで、現代のコンピューターでは複数のプログラムの実行を高速に切り替えることで、たくさんのプログラムを見かけ上同時に動かしています。これが、マルチタスクというしくみです。

さて、ジョブ制御やマルチタスクという概念によって、限られた計算資源を複数人で共有して利用できるようになりました。これは、コンピューターの利用効率を上げる利点がありましたが、同時に新たな問題を生み出しました。それは、アクセス権の分離されていた秘密にすべき情報が、限られた資源であるコンピューターを使うために一カ所に集まってしまう、ということです。

コンピューターを排他的に利用できた際には、秘密の情報をコンピューターで扱う際に、そもそもコンピューターの設置された部屋に許可された人以外を入れないようにすることで、情報にアクセスできる人を制限していました。実際、ENIAC の時代には、コンピューターの置かれた部屋自体に厳しい入室制限が課せられていました。しかし、コンピューターを共有して利用する場合には、少しのミスが命取りになりかねません。たとえば、学校の成績を処理するプログラムが、各生徒の点数をメモリ上に展開して計算したあとにそのデータを消さずにいたときに、その学校の科学部の生徒はプログラムを走らせることが許されていたら、悪意のあるプログラムを実行することで、全生徒の成績一覧を手に入れることができってしまうかもしれません。このように、コンピューターを共有することは、その上で扱う情報の機密性 (Confidentiality) を損なうことになりかねないのです<sup>注6</sup>。

これを解決するために生み出されたのが、権限 (privilege) や保護 (protection) という概念です。情報を確実に保護するためには、人間の性善説を信じるだけでは足りません。現代のコンピューターには、ある権限を持ったコードしか、あるデータを読んだり変更したりできないように、ハードウェア的に制限する機構が備わっています。この保護機構を活用すれば、資源の分離を司るコードに高い権限を割り当てつつ、より低い権限で信頼性の担保されていないプログラムを実行できます。これにより、万が一悪意のあるプログラムが実行されたとしても資源の分離を維持し、情報を守ることができるようになるのです。

このような保護機構が生まれたことで、「強い権限を持つプログラム」と「弱い権限を持つプログラム」に明確な境界線ができました。「弱い権限を持つプログラム」でも、「強い権限を持つプログラム」の提供するしくみを利用すれば、強い権限でしかできないこと、たとえばハードウェアの制御のようなことを代行してもらうことは可能です。……おや、そう言えばハードウェアの制御のようなコードは、複数のプログラムで共通になりがちな部分でしたよね。そう考えると、

---

注6 参考：『情報セキュリティの敗北史——脆弱性はどこから来たのか』（アンドリュー・スチュワート著、小林啓倫訳、白揚社、2022年）

強い権限を持つプログラムは、複数のプログラムから共通で利用される機能を提供したら都合が良さそうです。これなら、その共通の処理だけをしっかり書いておけば、あとは弱い権限で実行することで、お互いにリスクを避けつつ計算機を共有して利用できるのですから。

そういうわけで、現代のOSは「ハードウェアの抽象化」と「資源の分配」という機能を担当する部分として、アプリケーションとハードウェアの間にあるソフトウェアとして、次第に確立されてきたのです。

## 本書で実装するOSの全体像

OSとはどんなものか大まかにつかめたところで、本書が最終的に実装したいOSがどのようなものになるのか、その全体像を説明します。できる限りインクリメンタルに開発を進めていきたいので、最低限必要になる機能を最初に実装し、以後の章ではその時点で利用可能な機能を利用して、より複雑な機能を順次実装していきます。そのため、1回目から完全に理解できなくてもかまわないので、ページ順に最後までいったんは読み進めることをお勧めします（筆者も難しい本を読む際は「まだじっくりこないけれど、そういうもんなんだなあ」といったん飲み込んで最後まで読み進めるようにしています。その後にもう一度最初から読み直すと全体像が見えている分、初回よりも理解できる部分が格段に増えているはずです。ぜひ実践してみてください!）。

## ベアメタルプログラミング

次の第2章では、OSのない環境（ベアメタル、Bare-metal）でプログラムを動かす方法を学びます。

OSは、コンピューターで最初に起動するプログラムである、と教科書に書かれていることもあります。この表現は実は少し不正確です。コンピューターの電源を入れると多くの場合は、ハードウェアに内蔵されているファームウェアと呼ばれるソフトウェアが最初に起動します。そして、ファームウェアが何らかの方法でOSを外部の記憶装置からメモリ上にロードします。最後に、ロードされたOSに実行を移すことで、OSが起動されるのです。

本書では、UEFI というファームウェアが存在する x86 マシンを前提に OS を実装します。そのため、UEFI によってロードされ実行されるプログラムとして OS を書いていきます。この環境では、自分たちが OS (となるべき存在) です。また、UEFI の機能も、ある段階から使えない状態に入ります。このように、ハードウェアの提供する機能だけが利用できる環境のことを、ベアメタル環境と言います<sup>注7</sup>。

ベアメタル環境では、何をするにもすべて自分でやらなければいけません。このような過酷な環境でも生き残れるプログラムを書く方法を学び、OS という文明を築く基礎を構築していく章となります。

## 資源 (メモリと CPU) の管理

第 3 章と第 4 章では、OS の主要な役割の一つである、資源の管理に関わるしくみを実装します。コンピューターを構成する「資源」にはさまざまな種類がありますが、ここではその根幹をなすメモリと CPU に関係する部分を実装します。

第 3 章のテーマはメモリ管理です。どのような種類のソフトウェアでも、ソフトウェアがデータであり、データがメモリ上に配置される以上、ある程度のメモリを消費します。OS 自身もその例外ではありません。自分自身が利用するメモリを適切に管理し、無駄なく使えるようにする必要があります。これを達成するために、メモリの確保と解放を担当する、メモリアロケータというしくみを実装します。また、CPU の提供する資源管理と保護機構の例として、ページングや割り込み処理 (何らかのイベントに応じてプログラムの流れを強制的に変更するしくみ) についても扱います。

第 4 章では、マルチタスクを実装します。複数のソフトウェアを 1 つの CPU の上で同時に動かすこと、つまりマルチタスクを実現するためには、利用可能な CPU の時間を、各ソフトウェアに分配する必要があります。マルチタスクの実現方式はさまざまな種類がありますが、ここでは協調的なマルチタスクを、Rust の `Async/Await` や `Future` という機構を活用して実装します。

---

注7 この「ベア」は熊 (bear) ではなく裸足 (bare-foot) のベアです。

## ハードウェアの制御

第 5 ～ 6 章では、OS のもう一つの役割である、ハードウェアの制御・抽象化のうち、制御に関わる部分について実装します。ハードウェアの抽象化については、制御を実装するうえで自然と実装されるほか、本書のⅡ巻でシステムコールを実装する際により詳しく触れる予定です。

現代のコンピューターには、さまざまなハードウェアが搭載されています。そして、星の数ほど存在するハードウェアには、それぞれに異なる制御方法が存在します。そのすべてを説明し実装するには、人類の一生はあまりにも短すぎますし、現実的ではありません。そこで本書では、執筆時点でそこそこ広く用いられているハードウェアのうち、どうしても本書で紹介する OS に欲しいものをピックアップして、それらのしくみや制御方法、背景知識などを解説します。具体的には、現代のコンピューターで最も広く用いられているインタフェースである USB のコントローラを制御し、USB 接続のキーボードやマウスといった一般的なデバイスの制御についても紹介します。

## 本書のゴールと関連書籍の紹介

ここまでの章を終えれば、USB キーボードやマウスといったハードウェアの制御に加え、メモリという空間的資源の分配と、協調的なマルチタスクという時間的資源の分配を行うシステムソフトウェアができあがります。より具体的には、キーボードのどのキーが押されたのか OS が認識できるようになり、またマウスポインタが指している画面上の位置も検出できるようになります。

さらに、今後発売予定のⅡ巻では、Ⅰ巻で実装した内容をベースに、GUI (*Graphical User Interface*) やネットワークの実装を追加していきます。また、OS とは異なる独立したバイナリの実装を追加して実行する機構も追加します。これにより、既刊の姉妹書『[作って学ぶ] ブラウザのしくみ』で実装しているブラウザアプリケーション *saba* を動作させることができるようになります！

本書のⅠ、Ⅱ巻と『[作って学ぶ] ブラウザのしくみ』をあわせて読むことにより、現代の必須アプリケーションであるブラウザが動作するその裏側を、ブラウザの内部のみならず、OS の中身まで含めて、そのソフトウェアスタックを上

から下までみなさんの手で作り、そして理解できるようになるはずです。

## 本題に入る前に

本書を読み終えた暁には、OSとはどのようなもので、その構成要素にはどのようなものがあり、それらを実装する際はどのようなポイントに注目すればよいか、大まかに理解できるようになるはずです。一読目ではすべての内容を完全に理解することは難しいと思いますし、この本に書かれていることがOSのすべてというわけでもありません。しかし、本書で学んだことは、より深くOSについて学んだり、より良いOSの実装のアイデアを試してみたり、OSのことも視野に入れた効率の良いアプリケーションを開発したりする際の良い手助けとなると思います。ぜひ、ゆっくりでもよいので諦めずに読み進め、そして自分の手を動かしていろいろと試してみたり考えてみたりしてください。時間をかけて試行錯誤することが、理解への一番の近道であると筆者は考えています。

それでは、さっそくOSの世界に飛び込みましょう！