内田公太 [著]
@uchan_nos

マイナビ

下記のサイトにて本書のサポートを行います。質疑応答、正誤表の提供、読者同士のコミュニケーションを主な目的としています。

「ゼロからの OS 自作入門」サポートサイト：
https://zero.osdev.jp/

この本で作るMikanOS をビルドしたり、起動したりする方法（＝開発環境を整える方法）については、下記のGitHubリポジトリにまとめています。

https://github.com/uchan-nos/mikanos-build

Linux ディストリビューションの1 つであるUbuntu で開発するのが標準ですが、Windows のWSL上に用意したUbuntu を使っても開発可能です。詳細については本書「付録A」を参照してください。

マイナビ出版サポートページ：
https://book.mynavi.jp/supportsite/detail/9784839975869.html

- 本書は2021年2月段階での情報に基づいて執筆されています。本書に登場する製品やソフトウェア、サービスのバージョン、画面、機能、URL、製品のスペックなどの情報は、すべてその原稿執筆時点でのものです。執筆以降に変更されている可能性がありますので、ご了承ください。
- 本書に記載された内容は、情報の提供のみを目的としております。したがって、本書を用いての運用はすべてお客様自身の責任と判断において行ってください。
- 本書の制作にあたっては正確な記述につとめましたが、著者や出版社のいずれも、本書の内容に関してなんらかの保証をするものではなく、内容に関するいかなる運用結果についてもいっさいの責任を負いません。あらかじめご了承ください。
- 本書に記載されている会社名・製品名等は、一般に各社の登録商標または商標です。本文中では ©、®、™ 等の表示は省略しています。

はじめに

この本は **OS** を手作りする本です。OS 作りに関する知識が無いところから始めて、30 章で簡単な機能を持つ OS、名付けて「MikanOS」を作ります。30 章が終わった後には画面のようになります。この画面にあるものすべて、この本に書かれている内容を元に手作りしてるんですよ。もちろん、これらの機能は、他の OS の力を一切借りずに自力で動いています。

MikanOS の 30 日後の姿

この画面は MikanOS でいくつかアプリを起動させた様子です。左上では日本語の MikanOS 取扱説明書を表示しています。左下ではカラフルな立方体がくるくる回っています。中央には黒い背景のターミナルが起動していて、右下は富士山の JPEG 写真です。その他にもいくつか小物アプリが起動しています。右上には OS 起動時からの経過時間（0.01 秒刻み）を示すウィンドウと、その上にマウスポインタがありますね。全体的になかなか本格的な見た目じゃないでしょうか？（笑）

OS とはオペレーティングシステム（Operating System）の略です。OS はコンピュータの中で基本となるソフトウェアで、Windows、macOS、Linux などが有名です。人間がコンピュータを利用する際に OS は重要な役割を果たします。Web ブラウザやワープロなどのアプリが共通に利用する機能を提供することでアプリ開発を支援したり、計算資源を分配して複数のアプリを同時に使えるようにしたり、全体で統一した操作方法を提供したりすることで、OS はコンピュータを高度かつ使いやすくします。

Windows などの他人が作った OS の上で動くアプリではない、OS の力を借りずに動く「自作 OS」を作るのがこの本の目的です。他の OS の力を借りないということは、例えばマウスを操作したときに画面上のマウスポインタ（矢印）を移動させる処理を自分で書くということです。また、キーボードでコマンド入力し、Enter キーを押したらコマンドを起動する処理も自分で作ります。パソコンに搭載されたメモリ量を把握してメモリを管理する機能も作らないといけません。およそすべての処理を自分で作るのが「OS を手作りする」ということです。ワクワクしませんか？

OS 自作というのは一見するととっても無駄なことです。すでに高機能な OS がいくつも存在するのに、

それを後追いで作ろうというわけですからね。しかし、OS 自作は私たちに貴重な経験を与えてくれます。コンピュータシステムがどう動くのかを探求することは知的好奇心を大いに刺激します。実用面でも、パソコンのハードウェアや OS の処理内容に関する知識はソフトウェアエンジニアの仕事の幅を広げてくれます。OS の動作に目が向くようになり、効率的に動作するアプリの作り方を探求できるようになるでしょう。あるいはシステムの障害原因調査において、OS をデバッグした経験をもとに深いところまで原因追求ができるでしょう。いざ Linux カーネルのコードを読む必要に迫られても、OS を作った経験が有ると無いとでは読みやすさが格段に違います。

専門用語を使って MikanOS の特徴を紹介すると、MikanOS は UEFI BIOS により起動して 64 ビットモードで動作し、プリエンプティブマルチタスク、ウィンドウシステム、ページングによるメモリ管理、システムコールなどの機能を持った OS です。これらの専門用語の意味が分からなくても問題ありません。実際に物を作りながら意味を説明するのがこの本の役目ですからね。

対象読者

いまや個人で高機能な Web サービスや VR 対応のゲームを作れる時代になりました。IoT 向けの小型コンピュータを買えば、一昔前は想像できなかったような高度な電子工作をすぐ始められます。コンパイラや CPU の作り方を説明する本も出版されています。

いろいろなものが簡単に作れるようになりましたが、OS については逆に作りにくい時代になりました。コンピュータの高度化に対応して OS が進化した結果です。この本は、闇に隠されつつある OS 自作に光を当て、手作りの楽しさを途絶えさせないために書いています。筆者と一緒に、OS を手作りしてみませんか？

この本は 2006 年に出版された「30 日でできる！ OS 自作入門」（**参考文献 [1]**）の流れを汲んでいます。2021 年の今になっても絶版になっていない非常に素晴らしい本なのですが、出版当時からパソコンの中身は大きく進化し、内容の一部（とくにハードウェア関係）の情報は古くなり過ぎました。そこで当時の雰囲気を大事にしつつも、まったく新しい本を書き下ろすことにしたのです[＊1]。ということで、この本の内容は執筆時点で市販されているパソコンで試せるように工夫しています（ただし、すべての機種できちんと動くことを保証するわけではありません。その点はご了承ください）。

この本の対象読者は、簡単なプログラムを書いたことのある人を想定しています。数百行程度のプログラミング経験があると心配なく読み進められると思いますが、プログラムをちっとも書いたことがないと苦労すると思います。この本が初めてのプログラミング経験である、という方がいましたら、ぜひ筆者に読んだ感想を教えてくださいね。

筆者の趣味により、この本で作る MikanOS は C++ で書きました。C++ は C 言語を拡張して作られたプログラミング言語で、OS 作りにも活かせる便利な機能が豊富です。C++ にそれほど詳しくない方のために本文中で多少の説明を入れています。C++ をさらに詳しく勉強するには、この本で出てきた言葉を頼りに、入門書やウェブサイトを読んでみてください。もしかすると、OS 開発をする前に C++ を完璧に勉強しておこうと意気込むかもしれませんが、OS を作りながら必要に応じて勉強する方がやる気を維持しやすいのではないかと筆者は考えています。気張らずに前に進んでください[＊2]。

＊1　筆者が高校生のときに「30 日でできる！ OS 自作入門」の校正作業に参加しました。その当時、まさか、その流れを汲む本を執筆することになろうとは夢にも思っていませんでした。

＊2　編集注：「OS 自作」に興味を持ったら、ぜひ「第 0 章　OS って個人で作れるの？」も読んでみてください。

目次

第0章

OSって個人で作れるの？

皆さんが普段使っているであろう Windows、macOS、Linux はいずれも巨大な OS です。例えば Linux のバージョン 4.10（2017 年リリース）は約 2100 万行もあります。他の 2 つはソースコードが公開されていないので分かりませんが、数千万行はあると思います。もちろん、こんなに大きなソフトウェアを個人で作ることはできません。

それらの OS が最初から巨大だったかというとそうでもありません。Linux の作者リーナス・トーバルズ氏が作った最初のバージョン 0.01（1991 年リリース）は、たったの 1 万 239 行だそうです。バージョン 4.10 に比べるととても小さいですが、OS としての基本的な機能は備えていました。

また「30 日でできる！　OS 自作入門」で作成することになる「はりぼて OS」は 4249 行で、コンパイル後の大きさは 40KB 程度です。このくらいの大きさであれば個人で作れそうな気がしてきませんか？そう、作れちゃうのです！

本書で作る OS はシンプルなものですが、それでも GUI[*1] があってウィンドウがポコポコ表示され、マルチタスク[*2] で動くような、それなりにちゃんとしたものです。OS 本体は最終的に 1 万 1071 行になります。知っている人向けに書くと、UEFI で起動し、Intel 64 モードで動き、ページングを用いてメモリ管理を行い、USB3.0 ドライバを搭載し、ファイルシステムを持つような OS を作ります。

0.1 OS の作り方

「OS を作る」と言ったときに大きく分けて 2 通りの作り方があります。既存の OS を改造するか、全部自分で作るかです。

既存の OS、例えば Linux を基にして、必要な機能を足して不要な機能を取り除けばオリジナルな OS を作ることができます。実用的な OS を作るにはこのやり方が一番早いです。macOS や Android は基にした OS こそ違うものの、このやり方で作ってあります。ただこの方法では OS の全体を見渡すことは難しく、OS 作りのごく一部しか体験できないのです。せっかくなら全体をやってみたいと思いませんか？

本書では、OS を全部自分で作ってみることにします。パソコンの電源を入れて OS 本体を呼び出すところから、いろいろなアプリを動かせるようになるまで OS づくりを一通り体験します。読み進めるにつれて、今までブラックボックスだと思っていたパソコンの中身、OS の仕組みが分かってくるでしょう。

OS を作るにはコツがあります。それは、最初から完璧に作ろうとしないことです。最初から完璧を目指すと手が止まってしまって全然前に進まなくなります。そうではなく、最初は OS っぽく見えるおもちゃを作ることにしましょう。おもちゃを作りこんでいくと、最初はおもちゃだったものがだんだんと本物に近づいていくのです。

いざ OS を作るというと、やれモノリシックにするかマイクロカーネルにするかとか、メモリ管理はどうするんだとか、リアルタイム性は担保するのかとか、詳しい人からいろいろ言われるかもしれません。そういった質問にちゃんと答えるにはそれなりに勉強しなくてはいけません。OS 理論を学ぶことは、それはそれで大切なことです。

しかし、いろいろ勉強してからでなければ OS を書いてはいけない、なんてことはありません。むしろ、理論を知らないまま OS を作り始めた方が良いとさえ筆者は思います。なぜなら下手に勉強してしまうと先人が残したすごい理論に打ち負かされてしまい、OS 作りを楽しめなくなるからです（それで楽しめる人もいるとは思いますが）。OS 作りはとってもワクワクする創造的活動なのです。最初くらいは好き勝手に作って楽しむ方が、その後の勉強意欲も駆り立てられるというものです。

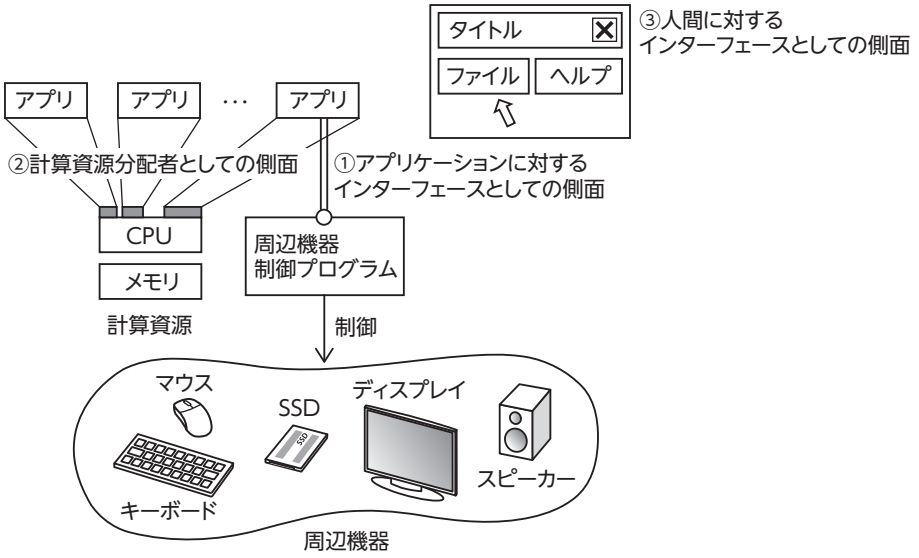
0.2 そもそもOSって何？

ここまで何となく OS という言葉を使ってきましたが、そもそも OS とは何でしょうか？　何を実現すれば OS と言えるのでしょうか？

実は人によって何を OS と呼ぶかはばらつきがあり、バシっと定義することは難しそうです。「30 日でできる！　OS 自作入門」にはこうあります。「さまざまな OS を比較してみたところ、この機能が共通点、といえそうなものを見つけられませんでした。結局のところ、それぞれの作者が『これは OS なんだ』と言い張って、周囲の人も『まあそうかな』と思えばどんなソフトでも OS なんです」。確かにそんな気もします。なぜなら、世界にはとてもシンプルな機能しか持たないけど「OS」と呼ばれるソフトウェアがたくさんあるからです。全部に共通する機能を探すのは大変です。

ただ、そこで思考停止せずに、よく使われる OS にどんな機能があるかを考えるのは有意義だと思います。すべての OS とまでは言わず、Windows、macOS、Linux など、読者の皆さんが普段使っているであろう OS を観察すると 3 つの側面が見えてきます（**図 0.1**）。

図 0.1　OS が持つ 3 つの側面



この図に登場するインターフェースとは 2 つの物の接合点を意味します。アプリと OS が接する点、あるいは人間とコンピュータが接する点ですね。日常生活で例えると、電子レンジや洗濯機のボタンは人間と機械が接する点ですのでインターフェースです。あるいは、家庭用ゲーム機のコントローラは人間とゲームをつなぐインターフェースです。細かく見ると、コントローラのボタンは人間に対するインターフェース、ゲーム機にあるコントローラ接続口はコントローラに対するインターフェースと言えま

＊1　Graphical User Interface。ボタンやチェックボックスのような図形パーツを用いた視覚的な操作方法。

＊2　複数のタスク（仕事）を同時に実行すること。今となってはマルチタスクは当たり前ですが、昔の OS にはマルチタスクができないものがありました。

す。

インターフェースは操作の窓口となります。インターフェースの良し悪しは製品の使いやすさに直結します。OS に当てはめると、アプリが OS の機能を使うときや人間がコンピュータを使うときに OS が提供するインターフェースを利用します。使いやすいインターフェースを提供できるかが OS 作者の腕の見せ所です。

コンピュータにはいろいろな周辺機器が接続されています。HDD や SSD などの 2 次記憶装置、マウスやキーボード、ディスプレイ、NIC[*3]、カメラ、スピーカーなどなど。OS が無かった時代は、各アプリがそれぞれの周辺機器を制御していました。OS は各アプリがやっていた周辺機器の制御を肩代わりします。通常、周辺機器（ハードウェア）の制御は複雑になりがちですが、その複雑さを OS が吸収してくれるため、アプリは単純化されたインターフェースを介して周辺機器を活用できます。

OS が提供する単純化されたインターフェースを使うことで、アプリは簡単に周辺機器を使えるようになります。加えて、アプリから周辺機器の詳細を隠すことで異なる（同種の）周辺機器を同じインターフェースで使えるようになります。例えばストレージ（HDD や SSD など）は製品により接続規格や速度が異なりますが、OS がその違いを吸収することで、アプリはファイルに対して read/write という単純化されたインターフェースを使いさえすればいいのです。このように、詳細を隠して単純なインターフェースを提供することを**抽象化**といいます。抽象化は本来複雑であるコンピュータの世界を人間の手に負える程度に簡単に保つために必須の仕組みです。

抽象化は、周辺機器の制御だけでなくネットワーク通信などにも応用されます。インターネット通信では TCP や UDP がよく使われますが、通常、その下にある伝送方法は OS に隠蔽されていてアプリからは見えません。もしかしたらマザーボードに接続された LAN ケーブルの Ethernet かもしれませんし、USB 経由で WiFi アンテナが刺さっているかもしれません。それぞれ、物理的には異なる伝送形式ですが、OS が抽象化したインターフェースを提供するためアプリは気にしなくて済みます。

今まで見たように「アプリケーションに対するインターフェースとしての側面」とは、プログラマがアプリを作る際にお世話になることが多い部分です。対して、これから説明する「計算資源分配者としての側面」は、アプリを使う側のお話です。

計算資源というのは CPU が持つ計算能力、メモリの記憶、ストレージの読み書き、NIC の送受信など、アプリの処理に必要なあらゆる資源（リソース）のことです。これらの資源は有限ですから、1 つのアプリが独占してしまうと他のアプリが実行できません。さらに、これらの資源を必要とするのはアプリだけではありません。マウスカーソルの移動処理など、OS が活動するためにも資源が必要です。複数のアプリや OS が並行に動けるよう、適切に計算資源を分配するのが OS の役目です。各資源をどう分割して分配するかは OS 作者の腕の見せ所です。

資源の分配は突き詰めると奥が深い話題です。例えば、動画のエンコードのような CPU をがんがん使うアプリを動かしている最中にマウスをちょっと動かしたとします。動画エンコードが終わるまでそのアプリが CPU を独占するような単純な作りの OS だとしたら、動画エンコードが終わるまでマウスが止まったままです。これでは、操作している人はパソコンがフリーズして壊れたと思うでしょう。動画のエンコードは CPU を大量に使うものの、一瞬なら処理が中断しても問題ありません。一方でマウスの移動処理は、平均すると CPU をほとんど使わないものの、マウスを動かした瞬間にはなるべく早く処理したいのです。普及している OS はどれも、このような場合でもちゃんと動くように工夫して作られています。

今まで見てきた 2 つの切り口は、主にプログラム的な話でした。プログラムを作りやすくしたり、複数のプログラムがうまく協調動作するようにしたり。最後の「人間に対するインターフェースとしての側面」は、人間がコンピュータを扱うのを OS が支援するというお話です。

パソコンを使うとき、各アプリに統一されたメニューがあると操作しやすいですよね。どのアプリにも同じような見た目でメインメニューがあり、「ファイル」メニューをクリックすると「上書き保存」や「名前を付けて保存」を選べる、という具合に。また、Ctrl-C でコピーして Ctrl-V で貼り付ける、というキーボードショートカットも、アプリ間で共通していてほしいですよね。このように、アプリ横断で共通のインターフェースを提供するのも OS[*4] の役割です。

3 つの切り口で OS の役割を見てきました。これらがすべて備わっていないと OS とは呼べない、というわけではありませんが、普及している OS にはそれらの機能が入っているということです。本書でも既存の OS にならい、多かれ少なかれそのような機能を持つように作っていこうと思います。

コラム 0.1 OS の仕様と POSIX

インターフェースというキーワードが出てきました。OS とアプリ、あるいは人間はインターフェースを介してやり取りします。そのため、インターフェースの利用方法や挙動がどのようなものであるか、つまりインターフェースの仕様が重要です。アプリからみたインターフェースの仕様といえば、例えば関数名や引数の型、関数の動作、戻り値などです。人間からみたインターフェースの仕様は、マウスをクリックしたときに何が起きるか、アプリの起動方法、アプリの結果をファイルに保存する方法、ウィンドウの「閉じる」ボタンの位置、などになりますね。OS のインターフェースがカバーする範囲は非常に広いです。

OS に限った話ではありませんが、インターフェースが同じであれば実装が異なっていても同じように利用できます。同じインターフェースに準拠する複数の OS があったとすると、1 つのアプリをどの OS でも動かせるようになり、アプリの作者や利用者は大変便利です。OS の作者にとってもこれは嬉しいことです。OS の対象ハードウェアを x86 パソコンにしようが、Arm スマホにしようが、自作 CPU を採用したコンピュータにしようが、共通のインターフェースさえ提供できれば既存のアプリを動かせることになりますからね。

OS のインターフェースとしては POSIX（Portable Operating System Interface）が有名です。これは UNIX 系の OS が広く採用しているインターフェースで、C 言語の関数やファイルシステム、プロセスなど、広い範囲をカバーします。本書で採用する標準 C ライブラリである Newlib は POSIX を前提として作られているため、MikanOS を作る際にも POSIX に関連した話がときどき登場します。

ただ、MikanOS 自体は既存のどのインターフェース仕様にも準拠しません。既存のインターフェース仕様を紹介するのが本書の目的ではないからです。むしろ、既存の仕様にとらわれず、作りたいものを作りやすい方法で実現しようと思います。POSIX やその他の既存インターフェース仕様と互換性のある OS を自作するのも非常に面白いと思いますから、読者の皆さんがそれに挑戦することは止めません。応援します！

＊3　Network Interface Card。ネットワークの送受信をする周辺機器。元々は拡張カードとして売られていたのでカードと呼ばれますが、現在はパソコン用マザーボードに標準で組み込まれています。

＊4　いやいや、それは OS（カーネル）ではなく GUI フレームワークの役目だ、という指摘はあり得ます。それはその通り。ここでは OS をカーネルに限定せず、もう少し広く捉えて説明しています。

0.3 OS自作の手順

普通のアプリケーションであれば、パソコンでソースコードを書いてコンパイルすれば実行可能ファイルが出来上がります。そのファイルを起動させれば（ダブルクリックするとか、ターミナルでファイル名を打ち込むとか）実行されますよね。Python などのインタプリタ言語であれば、ソースコードをそのまま実行することができます。

しかし、そうやって実行できるのも OS のお陰なのです。私たちが作ろうとしているのは OS そのものですから、他の OS に頼らずに動作する特殊なプログラムを作らねばなりません。次に示す手順で作っていきます。

1. **開発用パソコンで OS のソースコードを書き、コンパイルする**
2. **生成された実行ファイルを USB メモリに書き込む**
3. **試験用パソコンに USB メモリを接続し、電源を入れ、実行する**

開発用パソコンと試験用パソコンは同じでも構いませんが、その場合はいちいち再起動が必要となりますので面倒です。そこで、筆者は試験用パソコンとして GPD MicroPC という小型パソコンを使っています。5 万円程度で買えて機能が充実していますので、自作 OS を試験するにはぴったりです。そのほか、AMD の FX-8800P という CPU が搭載された A10N-8800E というマザーボードで組み立てた自作 PC も試験機としてたまに使っています。中古のパソコンを買ってもいいでしょう。2012 年以降に発売された機種であれば、本書の内容がそのまま動作すると思います。保証はできませんが。

試験用パソコンの代わりにエミュレータを使って自作 OS を試すこともできます。エミュレータというのは、パソコンの中に仮想的にパソコンを再現するソフトウェアです。エミュレータを使うことで、試験用パソコンを用意できない場合でも安心して本書の内容を試していただけます。エミュレータを使うと USB メモリの準備などをせず手軽に試せるので、試験用パソコンを持っていたとしてもエミュレータは活躍するはずです。筆者は MikanOS の開発のために QEMU というエミュレータをよく使っています。**「1.9　C 言語でハローワールド」**で QEMU を使った方法を説明します。

本書は Linux で開発を行う想定で書かれています。「30 日でできる！　OS 自作入門」は Windows で開発することになっていましたが、その時代に比べると Linux も十分使いやすくなりましたし、WSL[*5] という機能により Windows 上でも Linux が動きます。Windows よりも Linux の方が開発環境を整えやすいため[*6]、もしまだ Linux を使ったことがない方は、これを機に Linux へ入門してみてはいかがでしょうか？　正確に言えば Linux のディストリビューションの 1 種である Ubuntu 18.04 で動作を検証しています。WSL 上の Ubuntu と操作に違いがある場合は本文で説明しますので、Windows メインの方も一緒に楽しめます！

手順 1 でソースコードを書くテキストエディタは好きなもので構いません。筆者は Vim が好きですが、とにかく、最終的に目的の実行ファイルを得られれば道具は何でもいいのです。ソースコードを実行ファイルに変換するにはコンパイラというソフトウェアを使います。コンパイラにもいろいろありますが、本書では Clang（+LLVM）を使います[*7]。Clang のほかに GCC という有名なコンパイラがありまして、そちらに挑戦してみるのもまた面白いかもしれません。

手順 2 で使う USB メモリは、壊れても大丈夫なように安い USB メモリを買ってくることをお勧めします。容量は小さくて大丈夫です。また、試験用パソコンのファームウェアが対応しているメディアであれば USB メモリである必要はありません。内蔵や外付けの HDD や、ネットワーク経由の起動も可能です。筆者はいちいち USB メモリを抜き差しするのが嫌でしたので、USB メモリにはネットワーク経

由で OS を読み込むソフトウェア（iPXE）だけを入れておき、OS 本体のファイルは開発用パソコンに置いておく、というやり方で OS 開発をしています。こうしておけば、USB メモリはずっと挿したままで良くなります。

0.4 OS自作の楽しみ方

OS 自作に正解はありません。**「0.5　OS 自作の全体像」**では本書の流れを大まかに紹介しますが、単なる一例にすぎないのです。読者の皆さんはぜひ、迷うことを恐れず横道に逸れてみてください。独自の改造をしたり、本書で扱っていない機能を実装したりしてみてください。本書の流れをそのまま再現するだけでも OS や低レイヤについてそれなりの知識を得られると思いますが、横道を探検することでさらに奥深い経験を得られるでしょう。

本の内容を深く理解したい場合は「写経」をお勧めします。写経とはもともと、仏教の経典を書き写すことです。本書でいう写経は、サンプルコードをコピペせず自分で入力することを意味します。それに何の意味があるのだ、と思うかもしれません。コピペする方が圧倒的に素早く学習を進められますよね。

筆者の経験では、写経には細部を理解しながらコードを読めるという利点があります。コピペしたコードをただ眺めるより、1 文字ずつ自分で打ち込む方が細かい点に気付くからです。実際、筆者は「30 日でできる！　OS 自作入門」を写経しながら読み、深く理解できたと感じています。もちろん写経にはそれなりの時間がかかりますが、もしあなたが時間に余裕のある学習者であれば、写経にチャレンジしてみてはいかがでしょうか。

本書を読み進める中で内容について疑問が出てくるかもしれません。あるいは、独自の改造に挑戦しようとすると調べても分からないことがきっと湧いてくるでしょう。そんなときは筆者や、OS を自作している人のコミュニティに質問してみましょう。本書の内容に直接関連した質問であれば、本書のサポートサイト https://zero.osdev.jp/ が最適です。サポートサイトの中にある GitHub Issues で質問を投稿すると、筆者や他の読者の方が質問に答えます。また、本書の勉強メモを記録したり、独自の改造を発表したりする場として GitHub Wiki をご活用ください。サポートサイトには本書の読者が集まるわけですから、読書仲間を探すこともできます。ただし、GitHub の利用規約により 13 歳未満の読者の方は Issues や Wiki を使えませんので、後述する osdev-jp のメーリングリストに加入すると良いと思います。

本書で説明していない発展的な内容（例えば自作 OS から USB メモリを直接読み書きする方法など）に関する質問や議論は、筆者が運営している自作 OS コミュニティ「osdev-jp」が最適でしょう。osdev-jp にはパソコン用 OS だけでなく、組み込み機器向けの OS を作るようなメンバーもいます。

＊5　Windows Subsystem for Linux。Windows 10 に搭載されている機能で、Windows と Ubuntu などの各種 Linux ディストリビューションを共存させることができます。

＊6　大抵の Linux ディストリビューションには開発ツールが付属していたり、簡単にインストールできるようになっていたりします。本書の範囲を超えて、例えば Git を使ってソースコードのバージョン管理をしてみようとか、違うコンパイラを試してみよう等と思ったら、Linux の方が便利でしょう。

＊7　Clang を選んだ理由はいくつかあります。GCC より警告メッセージが読みやすいこと、オプションでビルドターゲットを指定できることなどです。ただ、これは好みの問題なので、GCC が好きな人はそちらを使っても良いです。

osdev-jp に加入する方法は https://osdev.jp/joinus.html を参照してください。osdev-jp のメーリングリストであれば、13 歳未満の方も加入できます。

本書で作る MikanOS は、すべてのパソコンで正常に動作するとは限りません。筆者は手元のパソコンで起動することを確認してはいますが、読者の皆さんが使っているパソコンで起動しない可能性は大いにあり得ます。そこが OS 開発の難しいところであり、また、楽しいところでもあります。アプリであれば、ハードウェアの些細な違いで動かなくなることは考えづらいですが、OS はハードウェアを直接扱いますから、個々のハードウェアの違いに大きく左右されるのです。もし MikanOS が動かない機種があったら、なぜ動かないのかを究明し、動くように修正を加えるのは皆さんに任されています。

0.5 OS自作の全体像

第 1 章からは具体的に OS の自作を進めていきます。かなり細かい内容を扱いますので、まるで森を探検するような感覚になると思います。ともすると森の中で迷ってしまうかもしれません。なるべく迷わないために、本編に入る前に全体を見渡しておきましょう。ただし、本編が濃密すぎて全体像を見渡すだけでもかなり長くなってしまいました。ごめんなさい。後で立ち返って来る際は目次が鳥瞰図[*8]としての役目を果たすでしょう。

第 1 章から第 3 章にかけて、ブートローダを作ります。ブートローダとは OS をメインメモリに読み込み、起動させるプログラムのことです。パソコンの仕組みからくる制約により、実行するプログラムはメインメモリに配置しなければなりません。しかし、一般的なメインメモリ（DDR-SDRAM）はパソコンの電源を落とすと内容が消えてしまいます。そこで、電源を切っても内容が消えないストレージ（HDD や SSD など）に OS を記録しておき、ブートローダを用いてストレージからメインメモリに読み出すのです。

本書では UEFI BIOS で動くブートローダを作ります。UEFI にはブートローダを作るための支援機能が沢山あります。例えば、UEFI はストレージを読み書きする機能を持っているため、ブートローダの作者が自分でストレージ装置の制御プログラム（デバイスドライバ）を作る必要がありません。UEFI の機能を呼び出すだけでいいのです。これで、かなり楽にブートローダを作ることができます。

ブートローダの後はいよいよ OS 作成を始めます。まずは図形や文字を画面に表示するのが目標です。パソコンの画面（ディスプレイやモニタと呼ぶ）はよーく見ると小さな四角（ピクセル）が縦横に沢山ならんだ構造になっていて、それぞれの小さな四角に任意の色を表示できます。適切なピクセルに色を塗っていけば図形や文字が表示できます。長方形や直線なら計算によって塗るべきピクセルの位置をはじき出せます。文字の場合はフォントという文字の形をデータ化したものを使って、どのピクセルを塗るかを決めます。ここまで来ると**図 0.2** のように、好きな色で長方形を描いたり、文字列を表示したりできるようになります。

＊8　鳥瞰（ちょうかん）とは鳥の視点のこと。高いところからは全体を見渡せるため、森の中での自分の居場所を把握しやすくなります。

図 0.2　フォント大集合

第 6 章、第 7 章ではマウスを使えるようにします。マウスが使えるようになるとマウスカーソルが画面内を移動できるようになります。これには割り込みという仕組みを使います。割り込みは、通常の処理に割り込んで処理を行う仕組みのことです。割り込みを使わずにマウスを制御しようとすると、一定間隔（例えば 0.01 秒ごと）で「マウスが動いたか？」というのをマウスに問い合わせる必要があります。平均してみるとマウスはほとんど止まっていて、問い合わせは無駄に終わります。かといって問い合わせ頻度を下げると「マウスがカクカクするなあ」と感じてしまうでしょう。割り込みを使えば、マウスが動いたときに割り込み処理が自動的に動作し、マウスカーソルが動くようになります。マウスを動かす処理は割り込みに任せるのが効率的なのです。

第 8 章ではメモリ管理の仕組みを作ります。メモリ管理とは、パソコンに搭載されているメインメモリの中で、どこが使用中でどこが空いているかを把握し、メモリを必要とするアプリやプログラムに対してメモリ領域を払い出すことです。そのためにはまず、メモリの容量と初期状態を知る必要があります。メモリが 4GB なのか 32GB なのかを知らなければなければなりませんし、OS 以外のプログラム（UEFI BIOS 自体やブートローダなど）が使っているメモリ領域を把握しなければ、正しいメモリ管理はできません。

その後に続く 2 章でウィンドウを表示できるようにします。ウィンドウの表示は文字の表示と同じで、基本的にはピクセルに絵を描くだけなのですが、重ね合わせの処理がちょっと難しいです。背景、ウィンドウ、マウスカーソルなどの重なりを考慮して描画しなければなりません。基本的には重なりのもっとも下にある絵から順に描くことで重ね合わせを表現すれば良いです。しかし、描画処理は大量のピクセルに色を描くため、単純なやり方では無駄に処理が重くなりがちです。マウスカーソルやウィンドウをスムーズに動かせるように高速化にも挑戦します。

第 11 章はタイマに対応します。タイマはパソコンに内蔵された時間を計るハードウェアのことです。時間を計れるということは一定時間毎に何らかの処理をするという仕組みが実現できるということです。複数のタスクを一定時間毎に切り替え、複数のタスクを並行に動かすプリエンプティブマルチタス

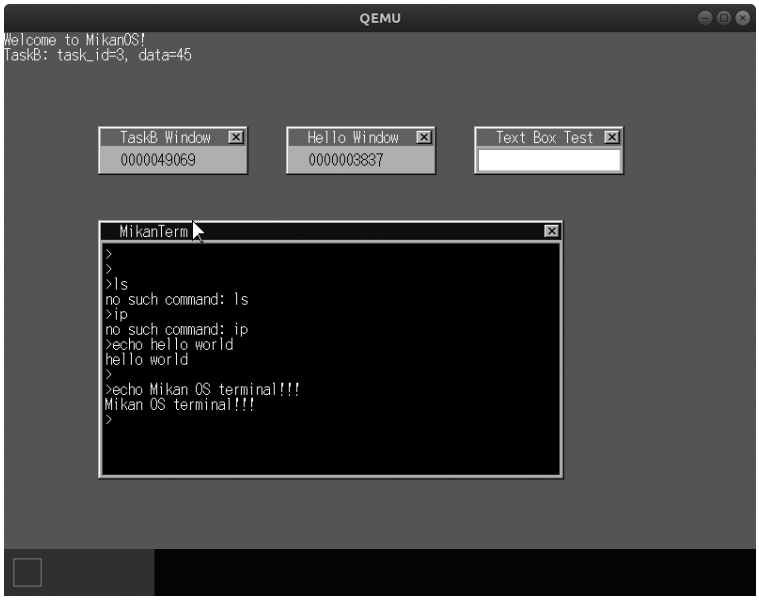
クの実現にはタイマが必須です。そのほか、ゲームや動画といった、時間をきちんと計る必要があるアプリケーションの実現にもタイマが活躍します。

第 12 章はキーボードに対応し、文字入力をできるようにします。キーボードの各キーにはキーコードと呼ばれる数値が振られていて、いずれかのキーを押すたびにキーコードが送られてきます。それを割り込み処理で受け取ります。キーをよく見ると複数の文字が印字されていますよね？　例えば日本語キーボードの「W」の上にある「2」キーには、2 のほかに「"」も印字されているはずです。キーコードはキーに対して振られた数値ですので、2 のつもりで押しても " のつもりで押しても同じ数値が送られてきます。OS は、Shift キーが同時に押されていたかどうかを考慮し、キーコードを適切な文字に変換します。

第 13 章、第 14 章ではマルチタスクという、複数のタスクを並行に動作させる仕組みに挑戦します。アプリはタスクの 1 種です。パソコンで作業するとき、音楽アプリで BGM を流しつつウェブブラウザで文献を調べ、ワープロアプリでレポートを書く、などと複数のアプリを同時に使いますよね。このようなことができるのはマルチタスクの仕組みのおかげなのです。マルチタスク機能がない OS では、1 つのアプリを閉じるまで他のアプリを使えません。

マルチタスクの仕組みは、パソコンの処理性能を最大限生かすためにも重要です。例えば秒間 10 フレームで動くゲームを作るとします。フレームとフレームの間は 0.1 秒だけ時間があり、1 フレーム目の処理が 0.02 秒で終わったとします。2 フレーム目までの 0.08 秒の空き時間を他のタスクに割り振ることができれば、その分だけ多くの処理をこなせます。人間は 0.08 秒の時間を与えられても何もできませんが、CPU はとても速いのでそのくらい短時間でも有効活用できるのです。マルチタスクの仕組みが無かったら、ほかにやるべき処理があるのにパソコンが暇な状態になってしまいます。

図 0.3　ターミナルの echo コマンドで遊ぶ様子



第 15 章、第 16 章ではターミナルとコマンドを実装します。ターミナルというのは黒い背景に白い文字が表示される、あの画面です。ターミナルは文字を使って命令したり結果を表示するための仕組みで

あり、CLI[*9]と呼ばれます。CLIでは、アプリは文字列だけを扱えば良いため、ウィンドウへのボタン配置やイベント駆動のプログラミングを必要とするGUIアプリに比較して、とても楽に作れます。ですので、特にOS開発初期にはターミナルはぴったりなのです。それに、ターミナルに命令を入力してエンターキーで実行するのはハッカーみたいでかっこいいですからね！　第16章の途中までいくと**図0.3**のような見た目になります。

第17章ではファイルシステムを作ります。現代のパソコンはバイトを基本にしてデータを扱います。バイトは0から255の数値を表現できる[*10]小さな記憶の単位です。英数字なら1バイトで1文字を表現できます。HDDやSSDなどのストレージは非常に大きなバイトの列だと思うことができます。どのくらい大きいかというと、1TBで広辞苑約2万冊分[*11]の文章を収録できます。2万冊の広辞苑が積まれた様子を思い浮かべてください。特定のページを探すのさえ大変ですよね。非常に扱いづらそうです。

ファイルシステムは、ストレージの広大なバイト列を小さく切り出し、「ファイル」として名前を付ける機能だと言えます。ファイルシステムは、ファイル名とバイト列の位置を対応付ける表を管理していて、名前で指定されたファイルがバイト列のどこにあるかを返したり、新規にファイルを作るときにバイト列の空き領域を探し、ファイル名と位置の対応表を更新したりします。バイト列の管理の仕方によって様々なファイルシステムがありますが、本書ではFATを採用します。

第18章から第20章にかけてアプリケーションを作れるようにします。今まではOS本体の機能を作っていましたが、ここからはOSの上で動くアプリを作っていきます。前節で紹介したOSの3つの切り口は、いずれもアプリがあってこそです。そもそも「アプリケーション」は「応用」という意味の英単語で、現実世界への応用、ユーザに何らかの価値をもたらすもの、という意味合いです。アプリがないOSは何の役にも立ちません。アプリが動くようになるのは、OS自作にとって大きな節目なのです。

特に第20章に作る**システムコール**の仕組みは、1つ目の切り口「アプリケーションに対するインターフェースとしての側面」を実現する大変重要な仕組みです。システムコールというのはアプリがOSの機能を呼び出すための窓口のことです。アプリはOSの様々な機能を活用して処理を進めていきますが、OSに含まれたプログラムを自由に呼び出せてしまってはセキュリティが保てません。自分で作ったアプリしか動かさないのであればいいですが、普通は第3者が作ったアプリを入手して使いますよね。もし悪い人が作ったアプリだったらシステム全体が乗っ取られ、他のアプリが持つ重要なデータを盗まれてしまいます。アプリとOSの接点をシステムコールに限定することで、セキュリティを保ったままOSの機能をアプリに提供できるようになります。OSを自作する大半の方は、他人のアプリを自分のOSで動かすなんてことはしないでしょう（してくれたら筆者としてはとても嬉しいですが！）から、システムコールなんてなくてもいいのです。でも、システムコールは一般的なOSには必ず備わる仕組みですので、本書でも作ってみようというわけです。

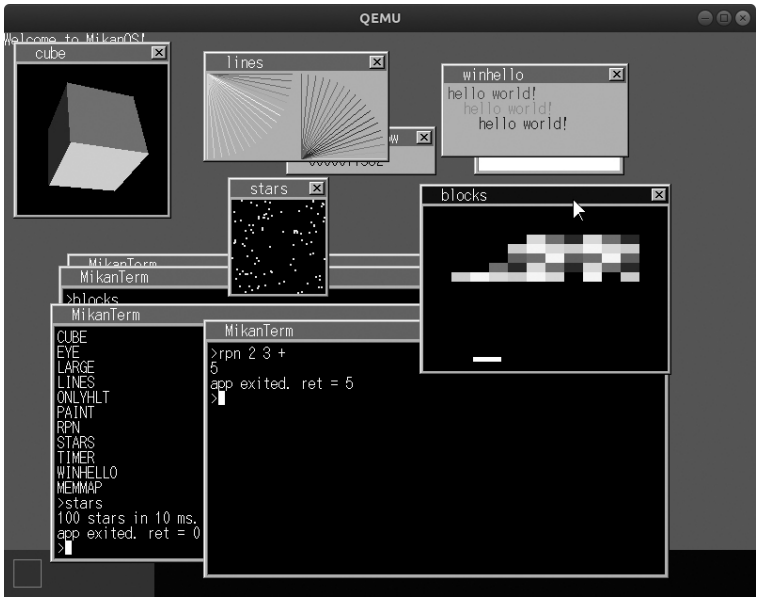
第21章から第23章は、ひたすらシステムコールを拡充していきます。ウィンドウを開いたり、絵を描いたり、時間を計ったり、キー入力したり。システムコールが増えるにつれ、作れるアプリの幅が広がっていきます。アプリをたくさん作るこの期間はとても楽しいと筆者は思います。OSの基礎的な仕組みを作るのも楽しいのですが、アプリが増えていくのは目に見えてモチベーションが上がりますよ。

＊9　Command Line Interface。コマンドラインとはターミナルに打ち込む命令のこと。CLIアプリというと、文字で命令し文字で結果を表示するようなアプリを意味します。

＊10　1バイトが何ビットであるかは決まっていませんが、本書で扱うx86-64アーキテクチャでは1バイト=8ビットです。8ビットで0から255までの数値を扱えます。

＊11　広辞苑1冊で約1500万字だそうです。1文字あたり平均3バイトだとすると1500万字で45MB。1TB/45MB ≒ 22222。ちなみに、原稿執筆時に家電量販店でSSDを調べると、500GBから1TB程度の製品が多く売られていました。

図 0.4　複数のアプリを起動した様子



第 24 章ではターミナルを複数起動できるようにします。本書で作る OS の設計では 1 ターミナル =1 アプリですので、複数のターミナルが開けるということは複数のアプリを同時に使えるということです。ここまで来ると**図 0.4** のような見た目になります。ずいぶん本格的な感じがしませんか？

第 25 章、第 26 章では、アプリがファイルを読み書きする仕組みを作ります。アプリはファイルディスクリプタ番号という整数値を使ってファイルを操作します。ファイルを開くと新たなファイルディスクリプタ番号が OS によって割り当てられ、以降アプリはその番号を使ってファイルの読み書きなどを行います。OS は番号とファイルディスクリプタの対応表を持ち、アプリが開いたファイルの一覧、それぞれのファイルの読み込み、書き込み位置を管理します。整数値でファイルを指定するというやり方が必須なわけではありませんが、広く採用されている方法ですので本書でも真似しました。同じ仕組みにしておけば、他の技術者と会話するときに話が通じやすいですからね。アプリからファイルの読み書きができるようになると、作れるアプリの幅が一気に広がります。

第 27 章ではアプリが大量のメモリを確保したり、ファイルをメモリに見せかけたりする仕組みを作ります。ここで作る仕組みは地味ですが高度で、「OS を裏から支える仕組みを作っているぞ！」という感覚になるでしょう。作れるアプリの幅はほとんど広がりませんが、技術的には興味深い内容となっています。

第 28 章の日本語表示は、OS の教科書としてはおまけみたいなものですが、でもやっぱり日本語に対応すると見た目が豪華になって楽しいので、ここで挑戦することにしました。日本語フォントは、本質的には英数字のフォントと同じです。文字数の多い日本語を表示するためにたくさんの文字の形が収録されているというだけのことです。FreeType というライブラリの力を借りて、TrueType フォントを使えるようにします。読者の皆さんの好きなフォントに変更して遊んでみてください。

第 29 章で挑戦するアプリ間通信[*12]（パイプ）は、複数のアプリが互いに情報交換するための仕組みです。今まで各アプリは単独で起動し、何らかの処理をして終了する、というものでした。アプリ間通信によって、あるアプリの出力結果を他のアプリに入力して処理させることができます。小さなアプリ

を組み合わせて複雑な仕事をさせられるようになり、一段とアプリの応用範囲が広がりますよ。

第 30 章は、おまけの機能やアプリを作ります。テキストファイルを大画面で見るためのテキストビューア、画像ファイルを表示する画像ビューアを作ると、画面が一気に華やかになります。3 枚の画像を表示した様子を**図 0.5** に示します。素敵でしょ？

図 0.5　画像ビューアで 3 つの画像を表示した

＊ 12　一般にはプロセス間通信と呼ばれます。