



オペレーティングシステム開発シリーズ

オペレーティングシステム開発 - はじめに

by Mike, 2009, Updated 2010

このシリーズは、オペレーティングシステムの開発を一から実演し、教えることを目的としています。

はじめに

Welcome!

本シリーズは、コンピュータとオペレーティングシステムに関する章、チュートリアル、記事をまとめたものです。このシリーズでは、アーキテクチャや、システムレベルのプログラミングにある概念を説明しながら、その過程でオペレーティングシステムを一から開発するという新しい方向性に焦点を当てています。

このシリーズの目的は、オペレーティングシステムとコンピュータシステムにおける最も包括的なガイドを提供することであり、その一方で、（シェアではなく）あらゆる点を網羅しようと試みています。

しかし、このシリーズを進める前に、選んだ言語について、読者が知っておくべきことを説明し、その言語の重要な概念の概要と、それをどのように扱うかを説明したいと思います。また、組み込みプラットフォームやシステムレベルのソフトウェアでしか使われない概念についても取り上げます。

このシリーズでは、**C言語**と**x86アセンブリ言語**を使用しています。先に進むためには、この**2つ**の言語をよく理解しておくことが非常に重要です。この章では、この**2つ**の言語の復習をします。

プログラムを組んだことがない人は・・・。

まず最初に、プログラミングとコンピュータサイエンスの世界に皆さんをお迎えしたいと思います。コンピュータ科学者やソフトウェアエンジニアは、プログラミングの手法を用いて複雑なソフトウェアシステムを構築します。プログラミングをしたことがない人は、最初からこの**3つ**を少しずつ学ぶことになります。そのため、最初のプログラミング言語を学ぶのは難しい傾向にあります。しかし、時間の経過とともに他のプログラミング言語の習得も容易になっていきます。

まずは**Python**や**Visual Basic**など、よりシンプルな言語から始めて、そこから発展させていくことをお勧めします。その後、**Java**や**C++**に移行するのがよいでしょう。**Java**は**C++**よりも簡単ですが、多くの構文を共有しているので、最初に**Java**を習得しておけば、**C++**への移行がより簡単になります。また、**C++**を学ぶことは、**C言語**の多くの部分を学ぶことになります。

以下のサイトをお勧めします。

- cprogramming.com
- cplusplus.com
- codecademy.com

最後に、**YouTube**には、ソフトウェア工学、コンピュータサイエンス、プログラミングに関する本当に優れたビデオや講義がたくさんあります。

- [プログラミング入門](#)
- [The Great Debate: どのプログラミング言語を最初に学ぶべきか?](#)

追加リンク

- [プログラミング言語を学ぶための初心者向けリソース タイトルブ](#)
- [ロのいじり方ガイド](#)
- [コンピュータプログラミング言語の歴史 プログラミング言語の初心者ガイド](#)

Cの概要

ここでは、C言語の重要な部分を簡単に説明し、それらがどのように機能するのかを説明します。

カーネルランドでのC言語の使い方

16ビットと32ビットのC

システムのプログラミングを始めてみると、助けになるものが全くないことに気づくでしょう。電源投入時、システムは32ビットコンパイラがサポートしていない16ビットリアルモードでも動作しています。16ビットリアルモードのOSを作ろうと思ったら、16ビットCコンパイラを使わなければならないということです。一方、32ビットのOSを作りたいと思ったら、32ビットのCコンパイラを使わなければなりません。16ビットのCコードと32ビットのCコードは互換性がありません。

このシリーズでは、32ビットのオペレーティングシステムを作成します。そのため、32ビットのCコンパイラを使用することになります。

C言語と実行ファイルのフォーマット

C言語の問題点は、フラット・バイナリ・プログラムの出力機能をサポートしていないことです。フラット・バイナリ・プログラムとは、基本的に、エントリー・ポイント・ルーチン（main()など）が常にプログラム・ファイルの最初のバイトにあるプログラムのことを指します。ちょっと待ってください。なぜそんなことが必要なの？

これは、古き良き時代のDOS COMプログラミングに通じるものがあります。DOS COMプログラムはフラットなバイナリで、明確に定義されたエントリーポイントもシンボル名も全くありませんでした。プログラムを実行するために必要なことは、プログラムの最初のバイトに「ジャンプ」することだけだった。フラット・バイナリ・プログラムは特別な内部フォーマットを持たないので、標準規格もなかった。そのため、ただ1と0が並んでいるだけだ。PCの電源を入れると、システムBIOS ROMが制御を行う。しかし、いざOSを起動しようとする、どうやって起動すればいいのかわからない。そのため、OSをロードするために、別のプログラム（ブートローダー）を実行します。BIOSは、このプログラムファイルがどのような内部形式で、何をやるものなのか全く知りません。そのため、Boot Loaderをフラットなバイナリプログラムとして扱います。ブートディスクのブートセクターにあるものは何でもロードし、そのプログラムファイルの1バイト目に「ジャンプ」します。

このため、ブートローダーの最初の部分（ブートコードやステージ1とも呼ばれる）をC言語で記述することはできません。これは、すべてのCコンパイラが、ライブラリファイル、オブジェクトファイル、実行ファイルなど、特殊な内部形式のプログラムファイルを出力するためです。これをネイティブにサポートしている言語は、アセンブリ言語しかありません。

ブートローダでのC言語の使い方

ブートローダの最初の部分がアセンブリ言語でなければならないのは事実ですが、ブートローダにC言語を使用することも可能です。これにはさまざまな方法があります。1つの方法は、Windowsと私たちのインハウスOSであるNeptuneの両方で使用されています。アセンブリのスタブプログラムとC言語のプログラムを1つのファイルにまとめます。アセンブリ・スタブ・プログラムは、システムをセットアップし、Cプログラムを呼び出します。これら2つのプログラムが1つのファイルにまとめられているので、ステージ1は1つのファイルをロードするだけで、スタブプログラムとCプログラムの両方をロードすることができます。

これは一つの方法ですが、他にもあります。GRUB や Neptunes のブートローダ、Microsoft の NTLDR や Boot Manager など、ほとんどの本物のブートローダは C 言語を使用しています。ここでは32ビットのC言語を使用しているので、32ビットのC言語に16ビットのコードを混在させることができる方法もあります。

これを実現するには、かなり複雑で厄介な作業になります。このため、シリーズのブートローダではアセンブリ言語の使用にとどめています。読者からの要望が多ければ、後にC言語を使った方法を説明する上級者向けのチュートリアルを行うかもしれません。

Cカーネルの呼び出し

ブートローダの準備が整うと、エントリーポイントルーチンを呼び出して、Cカーネルをロードして実行します。Cプログラムは特定の内部フォーマットに従っているため、ブートローダはファイルを解析し、エントリーポイントルーチンを呼び出すための方法を知っていなければなりません。シリーズでは、この方法を少し後に取り上げます。これにより、構築するカーネルやその他のライブラリにCを使用することができます。

ポインター

これを読んでくださっているあなたは、すでにポインタ^{ポインター}を使いこなしていると思います。システムソフトウェアでは、ポインタはいたるところで使われています。そのため、ポインタを使いこなすことは非常に重要です。

ポインタとは、簡単に言えば、何かのアドレスを保持する変数のことです。ポインタを定義するには、*演算子を使います。

```
char* pointer。
```

ポインタには「アドレス」が格納されていることを覚えていますか？上のポインタには何も設定していないので、どのような「アドレス」を参照しているのでしょうか？上のコードは、**ワイルドポインタ**の例です。ワイルドポインタとは、何でも指し示すことができるポインタのことです。**Cは何も初期化してくれないことを覚えておいてください。**このため、上のポインタは何でも指し示すことができます。他の変数、アドレス0、他のデータ、自分のコード、ハードウェアのアドレスなどです。

PAS (Physical Address Space) について

PAS (Physical Address Space) とは、使用可能なすべての「アドレス」を定義するものです。このアドレスは、PAS内のあらゆるものを指す。物理的なメモリ (RAM) やハードウェアデバイス、さらには何もない場所も含まれます。これは、WindowsのようなプロテクトモードのOSでプログラミングされたアプリケーションの場合、すべての「アドレス」がメモリであることと大きく異なる。

以下はその例です。アプリケーションプログラミングでは、以下のようにすると、セグメンテーションフォールトエラーが発生し、プログラムがクラッシュします。

```
char* pointer = 0;  
*pointer = 0;
```

これにより、ポインタが作成され、自分が「所有」していないメモリアドレス0を指すようになります。このため、システムはあなたがそこに書き込むことを許可しません。

さて、この同じコードを未来のCカーネルでもう一度試してみましょう。クラッシュするのではなく、**IVT (Interrupt Vector Table)** の1バイト目を上書きしています。

このことから、いくつかの重要な違いを知ることができます

- 。ヌルいポインタを使ってもシステムはクラッシュしない
- ポインタは、PAS内の任意の「アドレス」を指すことができ、それはメモリであってもなくても構いません。

存在しないメモリアドレスから読もうとすると、ゴミ（その時点でシステムデータバス上にあったもの）が出ます。存在しないメモリアドレスに書こうとしても、何もできません。存在しないメモリアドレスに書き込んで、すぐに読み返すと、「書き込んだ」だけの結果になることもあれば、ならないこともあります.....「書き込んだ」データがまだデータバス上にあるかどうかによります。

ここからが面白いところです。**ROM**デバイスは同じ**PAS**にマッピングされています。つまり、ポインタを使って**ROM**デバイスのある部分を読み書きすることができるのです。**ROM**デバイスの良い例は、システム**BIOS**です。**ROM**デバイスは読み取り専用なので、**ROM**デバイスへの書き込みは、存在しないメモリの場所へ書き込むのと同じ効果があります。しかし、**ROM**デバイスからの読み出しは可能です。

他のデバイスも**PAS**にマッピングされる可能性があります。これはお客様のシステム構成によって異なります。つまり、**PAS**の異なる部分を読み書きすると、異なる種類の結果が得られる可能性があります。

このように、ポインタは、アプリケーションプログラミングの世界ではなく、システムプログラミングの世界で大きな役割を果たしています。ポインタは、「メモリの場所を指す変数」ではなく、**RAM**であるかどうかに関わらず、「**PAS**内のアドレスを指す変数」と考えた方がわかりやすいかもしれません。

ダイナミック・メモリー・アロケーション

アプリケーション・プログラミングの世界では、通常、**malloc()**と**free()**、または**new**と**delete**を呼び出して、ヒープからメモリ・ブロックを割り当てます。これがシステムプログラミングの世界では違います。メモリを確保するには、次のようにします。

```
char* pointer = (char*)0x5000;
```

それです。いいでしょう？私たちはすべてをコントロールできるので、PASのどこかのアドレス（RAMでなければなりません）へのポインタを指定して、「1024バイトの新しいバッファがあります」などと言えいいのです。

ここで重要なのは、動的なメモリの割り当てがないことです。CやC++の動的メモリ確保はシステムサービスであり、OSが起動している必要がある。でも、待てよ！我々は独自のOSを開発しているのではないのか？そこが問題なのです :) malloc()やfree()、newやdeleteを提供できるようにするためには、独自のメモリ管理サービスやルーチンを書く必要があります。

それまでは、バッファを「割り当てる」には、アドレス空間の未使用の場所を使うしかありません。

Inline Assembly

C言語ではネイティブにできないこともあります。システムサービスやハードウェアデバイスとのやり取りには、アセンブリ言語を使用する必要があります。

ほとんどのコンパイラは、インラインアセンブリを可能にするキーワードを提供しています。例えば、Microsoft Visual C++では
_asm

```
_asm cli ; 割り込みを無効にする
```

また、アセンブリコードのブロックもあります。

標準ライブラリとランタイムライブラリ (RTL) について

外部ライブラリを使用することができますが、それらのルーチンがシステムサービスを使用していない場合に限り、printf()やscanf()、メモリルーチンなど、必要最低限のルーチン以外はすべて使用できます。ただし、その約90%はOSに合わせて書き換える必要があるので、自分で書くのが一番です。

RTLは、アプリケーションプログラムがランタイムに使用するサービスやルーチンのセットです。RTLは、その性質上、OSがすでに動作していて、それらをサポートしている必要があります。そのため、RTLは自分で開発する必要があります。

起動時のRTLコードは、C++のコンストラクタやデストラクタを呼び出す役割を担っています。C++を使いたい場合は、それをサポートするRTLコードを開発する必要があります。これにはコンパイラの拡張機能を使います。

シリーズでは、CやC++の機能をサポートするRTLと、必要に応じて基本的な標準ライブラリの両方を開発します。

エラーの修正

デバッグ

printf()もなければデバッガを使う方法もないため、何かがうまくいかないときにはどうすればいいのでしょうか？このシリーズでは、Bochsエミュレータに付属するデバッガであるBochsデバッガの使い方を紹介しています。このデバッガは、OSを動かすだけでなく、よくあるエラーを修正するのに使用できます。

唯一の方法は、情報を出力できるようなルーチンを自分で開発することです。せいぜい、ソフトウェアがクラッシュする前にどこまで到達したかを知ることができる程度でしょう。

次の機会まで

この章はここまでです。)次の章では、OSとは何か、そしてシリーズを通して使用するツールを紹介し、OSの世界への冒険を始めます。

次の機会まで。

マイク

BrokenThorn Entertainment 社。現在、DoE と [Neptune Operating System](#) を開発中です。質問やコメントはありますか？お気軽に [お問い合わせ](#) ください。

あなたも記事の改善に貢献したいと思いませんか？もしそうなら、ぜひ [私に教えてください](#)。

[ホーム](#)

第1章





オペレーティングシステム開発シリーズ

オペレーティングシステム開発 - はじめに

by Mike, 2008, Updated 2010

このシリーズは、オペレーティングシステムの開発を一から実演し、教えることを目的としています。

これは何のことでしょうか？

オペレーティングシステムは非常に複雑なテーマです。オペレーティング・システムの仕組みを学ぶことは、素晴らしい学習体験になります。

このシリーズの目的は、OS（オペレーティング・システム）開発のブラック・アートを一から教えることです。自分でOSを作ってみたい人も、単にOSの仕組みを知りたい人も、このシリーズはあなたのためのものです。

オペレーティング・システムとは？

オペレーティング・システムは、コンピュータの基本的な機能、外観、操作性を提供するものです。主な目的は、ユーザーにとって使いやすい操作環境を構築することです。

オペレーティングシステムの例としては、Windows、Linux、Macintoshなどがあります。

プログラムをしたことがない人は

コンピュータ・プログラミングとは、コンピュータが読み込んで実行するソフトウェア（プログラム）を設計・作成することです。しかし、OSにはこの機能が設計されている必要があります。

オペレーティング・システムとは、単一のプログラムではなく、相互に連携し、通信するソフトウェアの集合体です。これが「動作環境」の意味です。

OSはソフトウェアの集合体ですから、OSを開発するためには、ソフトウェアの開発方法を知らなければなりません。つまり、コンピュータ・プログラミングを知らなければならないのだ。

プログラミングをしたことがない方は、下記の「必要条件」をご覧ください。このセクションには、C++と80x86アセンブリ言語を使ったコンピュータ・プログラミングを学ぶのに役立つチュートリアルや記事へのリンクがあります。

要求事項

Cプログラミング言語の知識

C言語などの高級言語を使用することで、OSの開発が非常に容易になります。OSの開発でよく使われる言語は、C、C++、Perlです。しかし、これらの言語だけではなく、他の言語でも開発は可能です。FreeBASICを使ったものも見たことがあります。しかし、上位の言語を正しく動作させることは、長期的には作業を困難にすることもあります。

CとC++が一般的で、中でもCが最も多く使われています。Cは、中間レベルの言語であるため、高レベルの構成要素を提供しつつ、アセンブリ言語、つまりシステムに近い低レベルの詳細を提供しています。このため、OS開発においてC言語を使うことは非常に簡単です。これが、最も一般的に使われている主な理由の一つです。なぜなら、C言語はもともとシステムレベルや組み込みソフトウェアの開発のために設計された言語だからです。

このため、OSのほとんどはC言語を使用することになります。

C言語は複雑なプログラミング言語で、一冊の本ができるほどです。C言語を知らない方は、以下を参考に見てみてください。

- [LearnC- 初心者のための完全なリソース](#)
- cprogramming.com
- [C++で考える](#)

私は個人的に、今では廃れてしまったオリジナルの「The C++ Programming language」から学びましたが。

x86アセンブリ言語の知識

80x86のアセンブリ言語は、低レベルのプログラミング言語です。アセンブリ言語は、プロセッサの機械命令と直接1対1の関係を持つため、ハードウェアのプログラミングに適しています。

アセンブリ言語は低レベルであるため、C言語のような高レベル言語に比べて複雑で開発が難しい傾向があります。このため、必要なときだけアセンブリ言語を使用し、それ以上は使用しません。

アセンブリ言語も複雑な言語で、一冊の本が必要になるほどです。もしあなたがx86 Assembly Languageを知らないのであれば、以下が参考になるでしょう。

- [アセンブリ言語。ステップ・バイ・ステップの組立技術](#)

私は個人的に「Assembly Language Step by Step」（優れた入門書）と「Art of Assembly Language」から学びました。どちらもとても素晴らしい本です。

準備

それ以外のことは、私が途中で教えます。予めご了承ください。ここから先は、C言語やx86アセンブリ言語の概念を説明することはありません。しかし、**lgdt**や**sti**、**cli**、**bt**、**cpuid**などの使い慣れない新しい命令については説明します。

ツール・オブ・トレード

低レベルのコードを開発する際には、専用の低レベルソフトウェアが必要になります。これらのツールの中には、必要のないものもありますが、開発を大幅に支援することができるので、ぜひお勧めします。

NASM - アセンブラー

Netwide Assembler (NASM)はフラットバイナリの16bitプログラムを生成できますが、他のアセンブラ（Turbo Assembler (TASM)、Microsoft's Macro Assembler (MASM)）では生成できません。

OSの開発中、プログラムの中には純粋なバイナリ実行ファイルでなければならないものがあります。このため、NASMは使用するのに最適です。

NASMはこちらからダウンロードできます。

Microsoft Visual C++ 2005または2008

移植性を重視しているため、OSのコードのほとんどはC言語で開発されています。

OSの開発では、すべてのコンパイラがサポートしているわけではないが、コントロールしなければならないものがある。例えば、ランタイム・コンパイラのサポート（テンプレート、例外）や古き良き標準ライブラリに別れを告げることができます。また、システムの設計によっては、より詳細なプロパティをサポートまたは変更する必要があるかもしれません。例えば、特定のアドレスでロードする、プログラムのバイナリに独自の内部セクションを追加する、などです。）基本的な考え方は、世の中のすべてのコンパイラがOSコードを開発できるわけではないということです。

システムの開発には、Microsoft Visual C++を使用する予定です。しかし、DJGPPやGCC、あるいはCygwinなど、他のコンパイラで開発することも可能です。Cygwinは、Linuxのコマンドシェルをエミュレートするように設計されたコマンドシェルプログラムです。CygwinにはGCCの移植版があります。

Visual C++ 2008はこちらから入手できます。

また、Visual C++ 2005は今でも[ここ](#)から入手できます。

他のコンパイラへの対応

前述のように、他のコンパイラを使ってOSを開発することも可能です。ここでは主にVisual C++を使用していますが、お好みのコンパイラを使用できるように作業環境を整える方法を説明します。

現在、私は環境を整えることをテーマにしています。DJGPP

- Microsoft Visual Studio 2005
- GCC

また、可能であれば、以下のコンパイラをサポートするようにします。

- Mingw
- Pelles C

このリストにもっと追加したいという方は、[私にご連絡](#)ください。

ブートローダーのコピー

ブートローダは、512バイトの1セクタに格納される純粋なバイナリプログラムです。このプログラムなしではOSを作ることができないため、非常に重要なプログラムです。ブートローダは、BIOSによって直接読み込まれ、プロセッサによって直接実行される、OSの一番最初のプログラムです。

NASMを使ってプログラムを組み立てることはできますが、それをどうやってフロッピーディスクにするのですか？単にファイルをコピーするだけではだめです。その代わりに、Windowsが（ディスクをフォーマットした後に）作成するブートレコードを、私たちのブートローダで上書きしなければなりません。なぜそんなことをする必要があるのでしょうか？BIOSは、起動可能なディスクを探すときに、ブートセクタだけを見ていることを思い出してください。ブートセクタと「ブートレコード」は同じセクタにあります。そのため、上書きしなければなりません。

そのための方法はたくさんあります。ここでは2つの方法を紹介します。一方の方法でうまくいかない場合は、もう一方の方法をお試しく下さい。

警告以下のソフトウェアは、私が使用方法を説明するまでは、絶対に遊ばないでください。このソフトウェアを誤って使用すると、ディスクのデータが破損したり、PCが起動しなくなることがあります。

PartCopy - ローレベルのディスクコピー機

PartCopyは、あるドライブから別のドライブへのセクタのコピーを可能にします。PartCopyは "Partial copy" の略です。その機能は、ある場所から別の場所へ、特定のアドレスへ、特定の数のセクタをコピーすることです。

[こちら](#)からダウンロードできます。

Windows DEBUGコマンド

Windowsには、コマンドラインから使用できる小さなコマンドラインデバッガがあります。このソフトウェアでできることはたくさんありますが、私たちが必要としているのは、ブートローダーをディスクの最初の512バイトにコピーすることだけです。

コマンドプロンプトを開き、「debug」と入力します。小さなプロンプト(-)が表示されます。

```
C:\Documents and Settings\¥Michael>debug
```

ここにはコマンドを入力します。**h**はヘルプコマンド、**q**はquitコマンドを意味します。**w** (write) コマンドは、私たちにとって最も重要なものです。

例えば、ブートローダーのようなファイルをメモリにロードすることができます。

```
C:\Documents and Settings\¥Michael>debug boot_loader.bin
```

これにより、そのファイルに対して操作を行うことができます。(ファイルをロードするには、debugの**L(Load)**コマンドを使用することもできます)。上記の例では、**boot_loader.bin**が0x100番地にロードされます。

ファイルをディスクの第1セクターに書き込むためには、次のような形式の**W (Write)** コマンドを使う必要がある。

W [アドレス] [ドライブ] [ファーストセクター] [ナンバー]

なるほど...では、見てみましょう。ファイルはアドレス0x100にある。フロッピードライブ（ドライブ0）にしたい。最初のセクタはディスクの最初のセクタ（セクタ0）で、セクタ数はえっと...1です。

これをまとめると、フロッピーのブートセクターに**boot_loader.bin**を書き込むコマンドになります。

このコマンドの詳細は [Michael > debug - boot_loader.bin](#) をご覧ください。

VFD - バーチャルフロッピードライブ

フロッピードライブを持っていてもいなくても、このプログラムはとても便利です。保存されているフロッピーイメージから本物のフロッピードライブをシミュレートすることができますし、RAM上でも可能です。このプログラムは、仮想フロッピーイメージを作成し、フォーマットしたり、Windowsエクスプローラを使って直接ファイル（例えば、あなたのカーネルでしょうか？

[こちら](#)からダウンロードできます。

Bochs Emulator - PC用エミュレータおよびデバッガ

フロッピーディスクをコンピュータに挿入して、動作することを期待する。コンピュータを起動して、自分が作った最高の作品を見て感動する。...ブートローダでコントローラにコマンドを送るのを忘れたために、フロッピーのモーターが停止するまで。

低レベルのコードを扱う場合、注意しないとハードウェアを破壊してしまう可能性があります。また、OSのテストのために、開発中に何百回もコンピュータを再起動する必要があります。

また、コンピュータがただ再起動するだけの場合はどうしますか？カーネルがクラッシュした場合はどうしますか？OSにはデバッガがないので、デバッグは事実上不可能です。

解決策は？PCエミュレータです。VMWareとBochs Emulatorの2種類があります。私はBochsとMicrosoft Virtual PCを使ってテストしてみます。

Bochsは[こちら](#)からダウンロードできます。

以上、フォックスでした。

私が挙げたソフトウェアの使い方を知っている必要はありません。使い方は、使い始めてから説明します。

フロッピードライブのないコンピュータでシステムを動作させたい場合は、フロッピーイメージであってもCDから起動することができます。これは、ほとんどのBIOSがサポートしているフロッピーエミュレーションによって行われます。

フロッピーイメージからブート可能なISOを作成できるCD作成ソフト（個人的にはMagicISOを使っています）を入手してください。そして、そのISOイメージをCDに書き込むだけで、動作するはずです。

構築プロセス

上記のようにたくさんのツールがあります。それらがどのように役立つのかをよりよく理解するために、OSのビルドプロセス全体を見てみましょう。

- 全ての設定を行う
 1. VFDを使用して、使用する仮想フロッピーイメージを作成し、フォーマットします。
 2. Bochs Emulatorを設定し、フロッピーイメージから起動する。
- ブートローダ
 1. NASMでブートローダを組み立て、フラットなバイナリプログラムを作成します。

2. PartCopyまたはDEBUGコマンドを使って、ブートローダを仮想フロッピーイメージのブートセクターにコピーします。

- カーネル（そして基本的には他のすべてのプログラム

1. すべてのソースをオブジェクト形式（ELFやPEなど）にアセンブル、コンパイルし、ブートローダーで読み込んで実行できるようにする。

2. Windowsのエクスプローラーを使ってカーネルをフロッピーディスクにコピーする。

- テストしてみてください。

- 1.Bochsのエミュレータやデバッガを使ったり、本物のフロッピーディスクを使ったり、MagicISOを使ってブータブルCDを作成したり。

次の機会まで

ここに掲載されている用語や概念の中には、初めて耳にするものもあるかもしれません。これからの記事ですべて説明しますので、ご安心ください。

このチュートリアルのはじめの目的は、シリーズの残りの部分への足がかりを作ることです。基本的な紹介と、使用するツールのリストを提供します。これらのプログラムの使い方は、必要に応じて説明しますので、「必要条件」に記載されていること以外は、このチュートリアルを受ける必要はありません。

また、オペレーティングシステムを開発するための構築プロセスを見てみました。ほとんどの場合、かなり単純ですが、リストアップされたプログラムがいつ使用されるかを確認する方法を提供しています。

次のチュートリアルでは、最初のDOS（Disk Operating System）から時間をさかのぼり、歴史を少しだけ振り返ってみたいと思います。また、OSの基本的なコンセプトについてもご紹介します。

上記のツールはまだ使用しませんので、今すぐダウンロードする必要はありません。それでは、また次回。

マイク

BrokenThorn Entertainment社。現在、DoEと[NeptuneOperating System](#)を開発中です。質問やコメントは

ありますか？お気軽に[お問い合わせください](#)。

あなたも記事の改善に貢献したいと思いませんか？もしそうなら、ぜひ[私に教えてください](#)。



第0章

ホーム

第2章





オペレーティングシステム開発シリーズ

オペレーティングシステム開発 - 基礎理論

by Mike, 2009

このシリーズは、オペレーティングシステムの開発を一から実演し、教えることを目的としています。

はじめに

素晴らしい、クレイジーなオペレーティングシステムの世界へようこそ。

前回のチュートリアルでは、オペレーティングシステムとは何かを定義しました。オペレーティング・システムは、ユーザーとコンピュータ・システム間の基本的なインターフェースを提供します。システムの基本的なルック&フィールを提供します。

また、私たちを助けてくれる多くのツールを見てきました。アセンブラ、コンプライア、リンカー、PartCopy、MagicISO、Bochsなどです。

プログラミングの経験がない人がこれを読んでいたら（多少はあると思いますが）、恥ずかしいですね :) 最初のチュートリアルの "Prerequisites" のセクションを読み返してください。なぜまだ私を読んでいるのですか？頑張れ頑張れ

今回の記事では、これまでとは異なる方法でOSを見ていきます。まず、過去にさかのぼり、オペレーティングシステムの歴史を見てみましょう。これらのオペレーティングシステムには多くの類似点があることがわかります。これらの共通点は、オペレーティングシステムに共通する基本的な事柄に分類され、あなた自身のオペレーティングシステムを構築するための構成要素となります。

過去の出来事

現在のOSのほとんどはグラフィカルなものです。しかし、これらのグラフィカル・ユーザー・インターフェース（GUI）は、OSで実際に行われていることに対する大きな抽象化レイヤーを提供しています。

オペレーティングシステムの概念の多くは、プログラムがテープに記録されていた時代に遡ります。これらの概念の多くは、現在でも有効です。

前史 - オペレーティングシステムの必要性

1950年代以前は、すべてのプログラムはパンチカードに書かれていた。パンチカードは、コンピュータのハードウェアのすべての蛇口を制御する命令の形をしていました。それぞれのソフトウェアがシステムを完全にコントロールしていたのです。ほとんどの場合、ソフトウェアはお互いに全く異なるものであった。あるプログラムのバージョンでさえも。

問題は、それぞれのプログラムが全く異なることだった。常に一から書き直さなければならないため、単純なものにならざるを得なかったのである。また、ソフトウェアには共通のサポートがなかったため、ソフトウェアはハードウェアと直接通信しなければならなかった。これでは、移植性や互換性ありません。

メインフレームコンピュータの時代になると、コードライブラリの作成がより現実的になった。2つのバージョンのソフトが全く違うものになってしまうなどの問題は解消されたが、それぞれのソフトがハードウェアを完全にコントロールすることはできなかった。

新しいハードが出ればソフトも動かなくなる。もし、ソフトがクラッシュしたら、コントロールパネルからライトスイッチを使ってデバッグする必要がある。

ハードウェアとプログラムの間にインターフェースを設けるという発想は、メインフレームの時代に生まれました。ハードウェアに対する抽象化層を持つことで、プログラムは完全にコントロールする必要がなくなり、代わりにすべてのプログラムがハードウェアに対する単一の共通インターフェースを使用することになる。

この超クールなインターフェースは何でしょう？それは、私たちがオペレーティングシステムと呼んでいる、優しくてかわいい（時には厄介な）ものです。:)

1950年代 - 当時はOSがあった

ウィキペディアによると、最初に記録された本格的なOSは「GM-NAA I/O」。GM-NAA I/Oの後継機として登場したのがSHARE OSである。SHAREは、プログラムの共有、バッファの管理などを行い、アセンブリ言語で書かれたプログラムの実行を可能にした最初のOSである。SHAREは、1950年代後半にIBMコンピュータの標準OSとなった。



SOS (SHARE Operating System) は、バッファの管理、プログラムの共有、アセンブリ言語のプログラムの実行を可能にした最初のOSである。

"Managing Buffers" は、"Managing Memory" の一形態に関するものです。"プログラムの共有" は、異なるプログラムのライブラリを使用することに関連する。

ここで重要なのは、太古の昔から（実際にはそうではありませんが）、**OSはメモリ管理とプログラムの実行/管理を担当している**ということです。

これは私が説明している世界（コンピュータ）の歴史ではないので、古き良き時代のDOSに飛び移ることにしましょう。

1964年 - DOS/360とOS/360

DOS/360（または「DOS」）は、IBMが1964年の最後の日に発売すると発表したディスクオペレーティングシステムです。しかし、いくつかの問題が発生したため、IBMはDOS/360を3つのバージョンに分けて、それぞれ1966年6月に発売することになった。

そのバージョンは

- BOS/360 - 8KB コンフィグレーション。
- DOS/360 - ディスクで16KB構成。
- TOS/360 - 16KB テープ付きの構成。

また、DOS/360には**マルチタスク機能やメモリ保護機能がありませんでした**。同時期にIBM社が開発していたのがOS/360である。OS/360では「OS/MFT (Multiple Fixed Transactions)」を採用し、**ベースアドレスを固定**して複数のプログラムをサポートしていた。OS/360は「OS/MFT」(Multiple Fixed Transactions)を採用し、ベースアドレスを固定して複数のプログラムをサポートした。

これで、**マルチタスク、メモリ保護、固定ベースアドレス**などの興味深い言葉が出てきました。前に加えて、**プログラムの実行とメモリ管理**もあります。

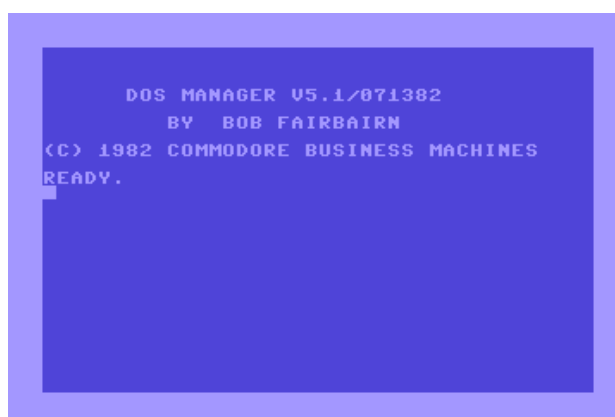
1969年、Unixだ!

C言語もUnixも元々はAT&Tが開発したものである。UnixとC言語は、政府機関や大学機関に自由に配布されたため、他のどのOSよりも幅広い種類のマシンに移植された。

Unixは、**マルチユーザー、マルチタスクのオペレーティングシステム**です。

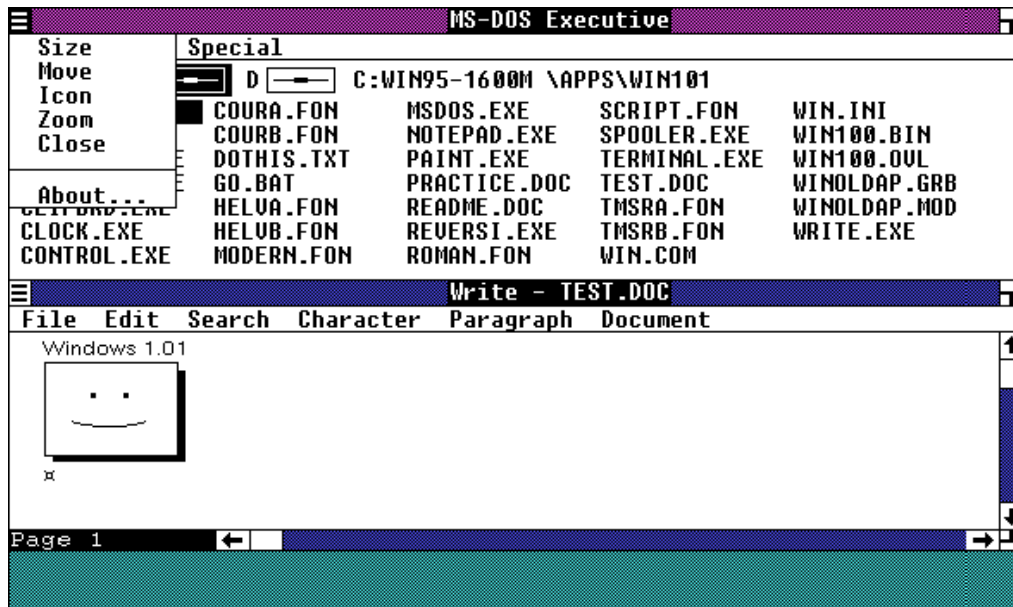
Unixには、**カーネル、ファイルシステム、コマンドシェル**があります。コマンドシェルを利用してOSと対話するGUI（グラフィカル・ユーザー・インターフェース）も数多く存在し、より親しみやすく美しい外観を実現しています。

1982年 - コモドールDOS



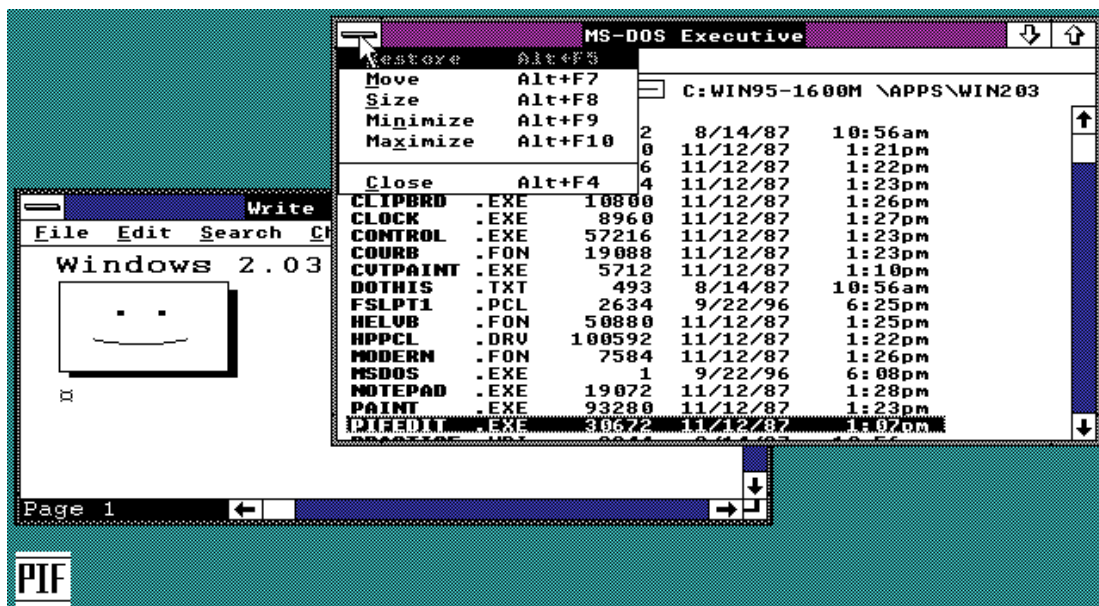
Commodore DOS (CBM DOS) は、Commodore社の8ビットコンピュータで使用された。起動時にディスクからシステムメモリにブートしていた前後のコンピュータとは異なり、CBM DOSはドライブ内のROMチップの内部で実行され、MOS 6502 CPUによって実行されていました。

1985年 - Microsoft Windows 1.0



初代Windowsは、DOSアプリケーションでした。その「MSDOS Executive」プログラムで、プログラムを実行することができます。しかし、どの「ウィンドウ」も重なることができなかったため、それぞれの「ウィンドウ」は横に並べて表示されていた。あまり人気はなかった。

1987年 Microsoft Windows 2.0



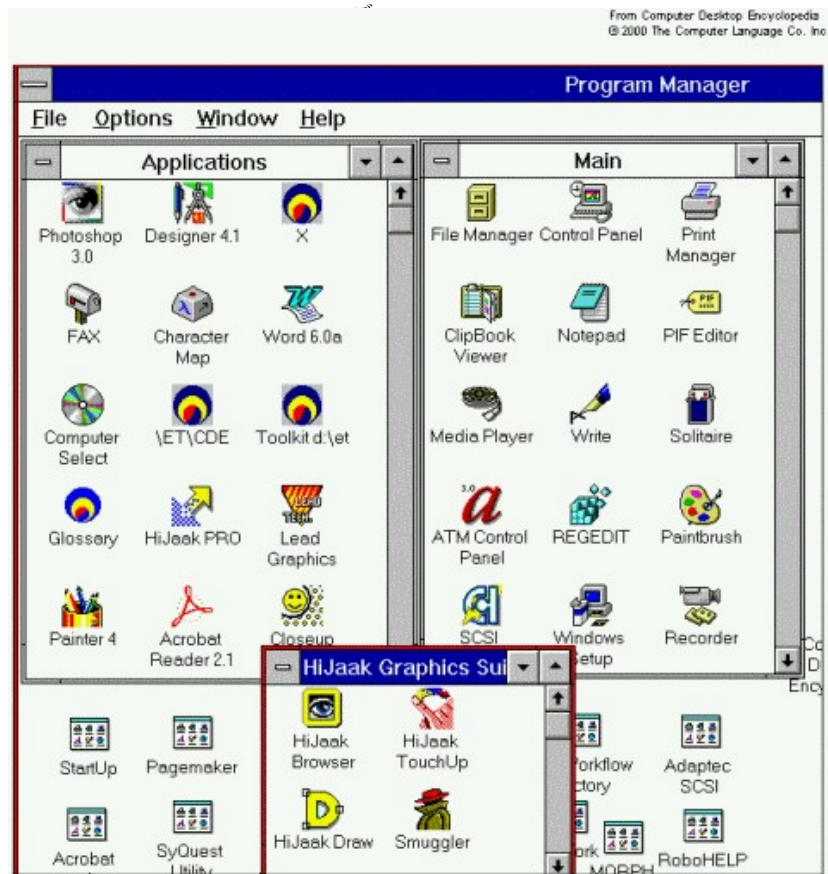
Windowsの第2版は、DOSのグラフィカルシェルでありながら、ウィンドウの重なりや色数の増加をサポートしていた。しかし、DOSの限界のため、あまり使われなかった。

注：DOSは16ビットのOSです。この時期、DOSはメモリをLinear Addressingで、ディスクをLBA (Linear Block Addressing) で参照しなければならませんでした。x86プラットフォームは下位互換性があるため、PCが起動すると16ビットモード (リアルモード) になり、LBAも残っています。これについては後ほど詳しく説明します。

16ビットモードの制限により、DOSは1MB以上のメモリにアクセスできませんでした。これは現在、キーボードコントローラで20番目のアドレスラインを有効にすることで解決しています。これについては後で説明します。

この1MBという制限のために、Windowsは圧倒的に遅く、それが不人気の第一の理由であった。

1987年 Microsoft Windows 3.0



Windows 2.0は完全に再設計された。Windows 3.0は、DOSのグラフィカルシェルであることに変わりはないが、DOSの1MBの制限を超えて16MBのメモリにアクセスできる「DOSエクステンダー」が含まれていた。これは、DOSプログラムとの**マルチタスク**をサポートします。

マイクロソフトを大きくしたWindowsです。サイズ変更可能なウィンドウや、移動可能なウィンドウをサポートしています。

OS開発者にとってのWindows

OS開発者の中には、次のWindowsを開発したいと考えている人が少なくありませんでした。それは可能ですが、非常に困難であり、一人のチームでは不可能です。上の図をもう一度見てください。これは、**コマンドシェル**の上に**グラフィカルなシェル**が乗っていて、それを**カーネル**が実行していることを思い出してください。また、Windowsもここから始めなければならなかったことを覚えておいてください。コマンドシェルがDOSで、グラフィカルシェルが「Windows」でした。

基本コンセプト

振り返ってみると、私たちの小さな記憶の旅は、いくつかの重要な新しい用語をもたらしました。これまでのところ、私たちは「オペレーティングシステム」に小さな定義を与えただけです。前のセクションでは、オペレーティングシステムとは何かについて、より良い、より説明的な定義をするのに役立つはずですが。

定義を明確にするために、上記の太字の用語をリストにしてみましょう。メモリ管理

- プログラムマネジメント
- マルチタスク
- メモリ
- 保護
- 固定ベースアド
- レス
- マルチユーザー
- カーネル
- ファイルシステム
- コ
- マンドシェル
- グラフィカル・ユーザー・インタ
- ーフェース (GUI)
- グラフィカル
- シェル
- リニア・ブロック・アドレッシング (LBA)
- ブートローダ (前回のチュートリアルより

考えることがたくさんありますね。しかし、上記のリストは技術的には抽象化されたレイヤーそのものです。も

う少し詳しく見てみましょうか。

メモリ管理

メモリマネジメントとは

メモリを要求するプログラムとの間で、動的にメモリを与えたり使用したりします。

-

- ページング、あるいは**仮想メモリ**のようなものを導入しています。
- OS カーネルが未知のメモリや無効なメモリを読み書きしないようにする。**Memory**
- **Fragmentation**の監視と処理。

プログラムマネジメント

これはメモリ管理と密接に関係しています。プログラム管理は以下の役割を担っています。プログ

- ラムが他のプログラムを上書きしないようにする。
- プログラムがシステムデータを破壊しないようにする。
- タスクを完了するためにプログラムからの要求を処理する（メモリの割り当てや解放など）。

マルチタスキング

マルチタスキングとは

- 複数のプログラムを切り替えて、一定の時間内に実行する。**タスクマネージャー**を用意し
- て切り替えを可能にする（Windowsのタスクマネージャーなど）。**TSS (Task State Segment)** 切り替え。またまた新語です
- 複数のプログラムをシミュレートして実行する。

メモリ保護

を参照しています。

- プロテクトモードで無効なディスクリプターにアクセスする（または無効なセグメントアド
- レスにアクセスする） プログラム自体を上書きする。
- メモリ上の他のファイルの一部または全部を上書きすること。

固定ベースアドレス

ベースアドレス」とは、プログラムがメモリ上に読み込まれる位置のことです。通常のアプリケーション・プログラミングでは、通常は必要ありません。しかし、OS開発では必要になります。

固定」ベースアドレスとは、簡単に言えば、プログラムがメモリにロードされるたびに、常に同じベースアドレスを持つことを意味します。プログラムの例としては、BIOSとブートローダがあります。

マルチユーザー

を指しています。

- ログインとセキュリティ保護
- 複数のユーザーがコンピュータ上で作業できること。データの損失
- や中断なしにユーザーを切り替えることができること。

カーネル

カーネルは、オペレーティングシステムの心臓部です。基本的な基盤、メモリ管理、ファイルシステム、プログラムの実行などの機能を提供します。近いうちにカーネルについて詳しく見ていきますので、ご安心ください。)

ファイルシステム

OS開発では、「ファイル」というものは存在しません。すべては、最初から（ブートセクタから）純粋なバイナリコードである可能性がある。

ファイルシステムとは、ファイルに関する情報を記述した仕様のことです。ほとんどの場合、クラスター、セグメント、セグメントアドレス、ルートディレクトリなどを参照しています。

ファイルシステムでは、ファイル名についても説明します。ファイル名には**外部**ファイル名と**内部**ファイル名がある。例えば、FAT12の仕様では、ファイル名は11文字までとなっている。それ以上でもそれ以下でもない。真面目な話です。つまり、例えば「KRNL.sys」というファイル名の場合、内部ファイル名は

“KRNL\$SYS”

ここではFAT12を使用しますが、その詳細については後述します。

コマンドシェル

コマンドシェルは、カーネルの上に独立したプログラムとして置かれます。コマンドシェルは、コマンドを入力することで、基本的な入出力を行います。コマンドシェルは、カーネルを使ってこの作業を支援し、低レベルのタスクを実行します。

グラフィカル・ユーザー・インターフェース (GUI)

GUI (Graphical User Interface) とは、簡単に言えば、グラフィカル・シェルとユーザーの間のグラフィカル・インターフェースとイン
www.broken Thorn.com/Resources/OSDev2.html

2021/11/15 13:03
タラクションのことです。

オペレーティングシステム開発シリーズ

グラフィカルシェル

グラフィカルシェルは、ビデオルーチンや低レベルのグラフィック機能を提供します。通常は、コマンドシェルで実行されます。(Windows 1.0,2.0,3.0のように)。しかし、最近は自動的に実行されるようになっています。

リニア・ブロック・アドレッシング (LBA)

オペレーティングシステムは、メモリ上のすべての小さなバイトを制御することができます。リニアアドレッシングとは、リニアなメモリに直接アクセスすること。例えば、以下のようになります。

```
mov     AX,                                OS開発におけるアクセス違反は存在しない
abl     [09000h].
e
```

これは良いことでもありますが、非常に悪いことでもあります。例えば

```
movable    bx, [07bffh] です           ; または7c00h以下の他のアドレス
movable    cx, 10
.loop1:
movable    [bx], 0h                    ; クリアbx
inc        bx                          ; 次のアドレスへ
ループ     .loop1                      ; cx=0になるまでループ
```

上記のコードは無害に見えます。しかし、もし上記のコードがブートローダの中にあった場合、上記のコードは10バイトだけ自分自身を上書きしてしまいます。痛い。理由は、ブートローダは 0x7c00:0 という固定アドレスで読み込まれており、上記コードは 07bffh から書き込みを開始しているからです。07c00hの1バイト前です。

ブートローダー

ブートローダです。この用語は前回のチュートリアルでも出てきました。前回のチュートリアルでは、ブートローダはBIOSによって読み込まれ、オペラシステムの最初のプログラムとして実行されることがわかりました。

ブートローダは、BIOSによって絶対アドレス0x7c00:0にロードされます。ロード後、CS:IPはあなたのブートローダに設定され、ブートローダが完全に制御します。

フロッピーセクターのサイズはわずか512バイトです。ブートローダは1つのブートセクタに収まらなければならないことを覚えておいてください。これは何を意味しているのでしょうか。ブートローダのサイズは非常に限られており、512バイトを超えることはできません。

ほとんどの場合、ブートローダはカーネルをロードして実行するだけのものか、**Second Stage Bootloader**のどちらか

です。近いうちに、ブートプロセスを詳しく見てみましょう。その時に、ブートローダについても見てみましょう。

結論

私たちは過去を振り返って、さらにいくつかの用語を学びました。歴史を学んだ後、私たちは用語を使って、すべてがどのように機能するのか、より幅広い視点を構築しました。いくつかのコードを見ることもできました。ほんの少しですが。

このようにして、私たちが何をしているのか、より簡潔に定義することができるはずです。

"ユーザーと供給されたプログラムのインターフェースを提供し、安定した安全な環境を提供するインタラクティブな環境、システムサービスとコンピュータハードウェアのインターフェース層"

うん。これが私の新しい "OS " の定義ですが、あなたはどうしますか？

次のチュートリアルでは、起動プロセスを詳しく見ていきます。その後、実際のブートローダの構築と組み立てを見てみましょう。

次の機会まで。

マイク

BrokenThorn Entertainment 社。現在、DoE と [Neptune Operating System](#) を開発中です。質問やコメント

はありますか？お気軽に [お問い合わせ](#) ください。

あなたも記事の改善に貢献したいと思いませんか？もしそうなら、ぜひ [私に教えてください](#)。



第1

ホーム

第3





オペレーティングシステム開発シリーズ

オペレーティングシステム開発 - ブートローダー by Mike, 2008, 2009

このシリーズは、オペレーティングシステムの開発を一から実演し、教えることを目的としています。

はじめに

ようこそ!このチュートリアルは、あなたが待ち望んでいたものです。以下のような多くのトピックを

- 取り上げます。ブートプロセス - その仕組み
- ブートローダ理論
- シンプルなブートローダの開発 NASMを
- 使ったブートローダの組み立て
- VFD(Virtual Floppy Drive)ソフトウェアの使用、PartCopyによるフロッピー
- イメージの作成、ブートローダーのフロッピーイメージへのコピー
- Bochsを使う - 基本的な設定と使用方法; ブートローダーのテスト

いいですか?

ブートプロセス

電源ボタンを押す

電源ボタンを押すと、実際に何が起こるのでしょうか?このボタンが押されると、ボタンに接続されたワイヤーがマザーボードに電子信号を送ります。マザーボードは、この信号を電源装置 (PSU) に向けて迂回させるだけです。

この信号には、1ビットのデータが含まれています。0であれば、もちろん電源が入っていません (つまり、コンピュータの電源が切れているか、マザーボードが死んでいる)。1であれば (アクティブな信号であることを意味します)、電源が供給されていることを意味します。

これを理解するには、コンピュータの2進法の基本を思い出してください。8ビットとは、簡単に言えば、電気が通る8本の "線" を表しています。0は電流がないことを表し、「1」は線の中に電流があることを表します。これが「ロジックゲート」と並んで、コンピュータの基礎となる「デジタル・ロジック・エレクトロニクス」の基本である。

PSUはこのアクティブな信号を受け取ると、システムの他の部分への電力供給を開始します。すべての機器に適切な量の電力が供給されていれば、PSUは大きな問題なく電力を供給し続けることができます。

そしてPSUは、「power_good」信号と呼ばれる信号をマザーボード内のBIOS (Basic Input Output System) に送ります。

BIOS POST

この「power_good」信号を受信すると、BIOSはPOST (Power On Self Test) と呼ばれるプロセスの初期化を開始します。POSTでは、十分な電力が供給されているかどうか、インストールされているデバイス (キーボード、マウス、USB、シリアルポートなど) が正常かどうか、メモリが正常かどうかをテストします (メモリの破損をテストします)。

その後、POSTはBIOSに制御を委ねます。POSTは、メモリの最後 (0xFFFFF0かもしれません) にBIOSをロードし、メモリの最初のバイトにジャンプ命令を置きます。

プロセッサの命令ポインタ (CS:IP)は0に設定され、プロセッサが制御を行います。

これは何を意味するのでしょうか?プロセッサは、アドレス0x0から命令の実行を開始します。この場合は、POSTが配置したジャンプ命令です。このジャンプ命令は、0xFFFFF0 (またはBIOSがロードされた場所) にジャンプし、プロセッサはBIOSの実行を開始します。

BIOSが制御する...

BIOS

BIOS (Basic Input Output System) はいくつかの機能を持っています。IVT (Interrupt Vector Table) を作成し、いくつかの基本的な割り込みサービスを提供します。さらに、BIOSはハードウェアに問題がないことを確認するためにいくつかのテストを行います。また、BIOSはセットアップユーティリティを提供します。

その後、BIOSはOSを探す必要があります。BIOSセットアップで設定した起動順序に基づいて、BIOSは割り込み (INT) 0x19を実行し、起動可能なデバイスを見つけようとします。

起動可能なデバイスが見つからない場合 (INT 0x19が返ってくる)、BIOSは起動順に記載されている次のデバイスに進みます。それ以上のデバイスがない場合は、「No Operating System found」に似たエラーを表示し、システムを停止します。

割り込みとIVT (Interrupt Vector Table) について

割り込みとは、さまざまなプログラムから実行可能なサブルーチンです。これらの割り込みは、アドレス0x0でInterrupt Vector Tableと呼ばれるテーブルに格納されています。一般的な割り込みは、例えば、DOSで使用するINT 0x21です。

注: DOSはありません。利用可能な割り込みは、BIOSで設定されている割り込みのみです。他の割り込みを使用すると、システムが存在しないルーチンを実行することになり、プログラムがクラッシュします。

注: プロセッサモードを切り替えると、IVTは利用できなくなります。つまり、ソフトウェアもハードウェアも、BIOSも含めて、一切の割り込みができなくなります。32ビットのOSでは、これをやらなければならないでしょう。でも、まだです。

BIOSインターラプト 0x19

INT 0x19 - SYSTEM: BOOTSTRAP LOADER

メモリのクリアやIVT(Interrupt Vector Table)の復元を行わずに、**Warm Reboot**でシステムを再起動することができます。

この割り込みは、BIOSが実行します。1枚目のハードディスクの第1セクター（セクター1、ヘッド0、トラック0）を読み込みます。

セクター

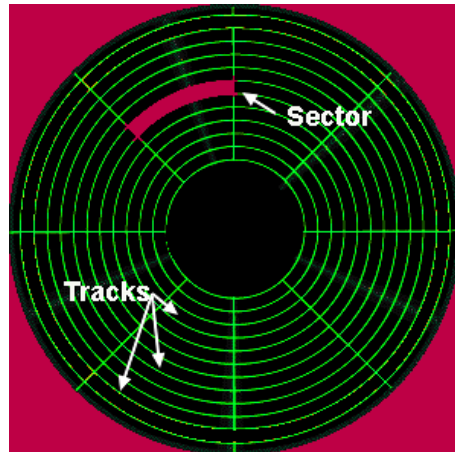
セクター」とは、簡単に言えば、**512バイト**のグループを表します。つまり、セクター1は、ディスクの最初の**512バイト**を表しています。

ヘッド

ヘッド」（またはフェース）は、ディスクの側面を表しています。ヘッド0が表側、ヘッド1が裏側になります。ほとんどのディスクは**1面**しかないので、ヘッドも**1つ**しかありません（「ヘッド1」）。

トラック

トラックを理解するには、絵を見てみましょう。



この写真では、このディスクは、ハードディスクやフロッピーディスクを表しています。ここでは、ヘッド1（表側）を見ており、セクタは**512バイト**を表しています。トラックは、セクタの集合体です。

注：1セクターは512バイトで、フロッピーディスクの1トラックは18セクターであることを覚えておいてください。これは、ファイルを読み込むときに重要になります。

ディスクが起動可能であれば、**0x7C00**にブートセクタがロードされ、INT0x19がそこにジャンプすることで、ブートローダに制御権が与えられます。

注：ブートローダは0x7C00にロードされることを覚えておいてください。これは重要なことです。

注：一部のシステムでは、0x0040:0072のアドレスに0x1234を置き、0xFFFF:0にジャンプすることで、ウォームブートを実行することもできます。コールドリブートの場合は、代わりに0x0を格納します。

これで、1337のブートローダが制御できるようになりました。

ブートローダ理論

これまでブートローダについて多くの話をしてきました。大事なところをまとめてみましょうか

。これまで、ブートローダは...

- ...マスターブートレコード (MBR) に格納されています。
- ...ディスクの第1セクターにあります。
- ...1セクター (512) バイトのサイズです。
- ...BIOS INT 0x19でアドレス0x7C00にロードされます。

さすがに**512バイト**では多くのことができません。では、どうすればいいのか。

アセンブリ言語では、**512バイト**を簡単に超えることができます。つまり、コードの見た目は問題なくても、その**一部**だけがメモリ上に存在することになるのです。たとえば、こんなコインロッカー。

```
movab    AX, 4CH
le
inc      bx          512バイト
movab    [var], bx   514バイト
le
```

アセンブリ言語では、ファイルの先頭から下に向かって実行を開始します。ただし、ファイルをメモリに読み込む際には、セクタを読み込むことを忘れてはならない。このセクタの1つ1つは**512バイト**なので、ファイルの**512バイト**だけをメモリにコピーすることになる。

上記のコードが実行され、最初のセクタのみがメモリにロードされた場合、**512バイト**（**inc bx**命令）までしかコピーされません。つまり、最後のmov命令はディスク上に残っていますが、**メモリ上には残っていないのです。**

inc bxの後、プロセッサは何をしますか？まだ**514バイト**へと進みます。このバイトはメモリ上になかったので、**ファイルの最後まで実行されます。**その結果は？クラッシュです。

しかし、第2セクタ（またはそれ以上）を所定のアドレスにロードして実行することは可能です。そうすれば、ファイルの残りの部分がメモリに入り、すべてがうまく機能するようになる。

2021/11/15 13:03

オペレーティングシステム開発シリー

このアプローチはうまくいきますが、ハードハックされてしまいます。最も一般的な方法は、ブートローダのサイズを 512 バイトにしておき、第 2 段階のブートローダを検索してロードし、実行するというものです。これについては後ほど詳しく説明します。

ハードウェアの例外

ハードウェア例外はソフトウェア例外と同じですが、ソフトウェアではなく**プロセッサ**が実行します。

すべての例外を発生させないようにしなければならない場合があります。たとえば、コンピュータのモードを切り替えるときには、割り込みベクトルテーブル全体が利用できないため、ハードウェアでも**ソフトウェアでも、割り込みが発生するとシステムがクラッシュしてしまいます**。これについては後で詳しく説明します。

CLIとSTIの説明

STIおよびCLI命令を使用して、すべての割り込みを有効または無効にすることができます。ほとんどのシステムでは、大きな問題を引き起こす可能性があるため、アプリケーションに対してこれらの命令を許可していません（ただし、システムはこれらをエミュレートすることができます）。

```
クラッシュ          割り込みの解除
何となくしてくれ...
sti                 割り込みを有効にすることで、問題は解決します。
```

ダブルフォールト・ハードウェア・エクセプション

実行中に問題を発見した場合（無効な命令や0による除算など）、プロセッサは割り込み0x8である**Second Fault Exception Handler（Double Fault）**を実行します。

ダブルフォールトについては後述します。ダブルフォールトの後にまだ続行できない場合、プロセッサは**トリプルフォールト**を実行します。

トリプルフォールト

この言葉、どこかで見たことがありますか？トリプルフォールト」になったCPUは、単純にシステムがハードリブートしたことを意味します。

ブートローダなどの**初期段階**では、コードにバグがあると必ずトリプルフォールトになります。これは、あなたのコードに問題があることを示しています。

シンプルなブートローダの開発

Yiipeel!*drum rolls* 待ちに待った瞬間です!:)では、もう一度リストを見てみましょう。

- マスターブートレコード（MBR）に格納されている
- ディスクの第1セクタにある。
- 1セクタ（512）バイトのサイズです。
- 0x7C00番地のBIOS INT 0x19でロードされます。

普通のテキストエディタ（私はVisual Studio 2005を使っています）を開きますが、メモ帳でも十分です。ブートローダ(Boot1.asm)ができあがりました。

```
;*****
;Boot1.asm
; - シンプルなブートローダ
;
オペレーティングシステム開発チュートリアル
;*****

オーガ          0x7c00          0x7C00でBIOSによりロードされます。
ビット    16          私たちはまだ16ビットのリアルモードにいます。
スタート。

        ク          すべてのインターラプトをクリア
        ラ          システムを停止させる
回 510 - 1($-$$) db 0          ; 我々は512バイトでなければならない。残りのバイトを0でクリア
dw 0xAA55          ブートシグニチャー
        チ
        ヤ
        チ
```

驚くべきことではありません。一行ずつ分析してみましょう。

```
オー          0x7c00          0x7C00でBIOSによりロードされます。
ガ
```

覚えておいてください。**BIOSでは0x7C00でロードされています**。上記のコードは、NASMに、すべてのアドレスが0x7C00に相対するように指示してい

ます。つまり、最初の命令は0x7C00になるということです。

ズ

ビット 16

私たちはまだ16ビットのリアルモードにいます。

チュートリアル2を覚えていますか？このチュートリアルでは、x86ファミリーが古いDOSシステムと下位互換であることを説明しました。昔のDOSシステムは16ビットでしたので、**x86互換のコンピュータはすべて16ビットモードで起動します**。これはつまり

- メモリは1MB(+64k)に制限されています。

コンピュータを32ビットモードに切り替える必要があります。この作業は後ほど行います。

```
回 510 - ($-$$) db 0
```

```
;我々は512バイトでなければならない。残りのバイトを0でクリア
```

これがもっと文書化されていればいいのですが。NASMでは、ドル演算子(\$)は現在の行のアドレスを表します。\$\$は最初の命令のアドレスを表します（0x7C00のはず）。つまり、\$-\$\$は、現在の行から先頭までのバイト数を返します（この場合、プログラムのサイズです）。

```
dw 0xAA55
```

ブートシグニチャー

これには説明が必要です。

BIOSのINT 0x19は、起動可能なディスクを検索することを覚えておいてください。ディスクが起動可能かどうかはどうやって判断するのでしょうか？ブートシグニチャーです。511バイト目が0xAA、512バイト目が0x55の場合、INT 0x19はブートルードをロードして実行します。

ブートシグニチャーはブートセクターの最後の2バイトでなければならないため、512バイト目ではなく510バイト目までを埋めるために、timesキーワードを使って異なるサイズを計算します。

NASMで組み立てる

NASMはコマンドラインアセンブラであるため、コマンドラインまたはバッチスクリプトで実行する必要があります。Boot1.asmをアセンブルするには次のようにします。

```
nasm -f bin Boot1.asm -o Boot1.bin
```

fオプションは、NASMにどのようなタイプの出力を生成するかを伝えるために使用されます。この場合は、バイナリプログラムとなります。

-o オプションは、生成されたファイルに別の出力名を付けるために使用します。この例では、Boot1.binです。

アセンブル後、"Boot1.bin"という正確な512バイトのファイルができるはずですが。

注：Windowsのエクスプローラーでは、何らかの理由でファイルサイズの表示が1KBに制限されています。ファイルのプロパティを見ると、512バイトと表示されているはずですが。

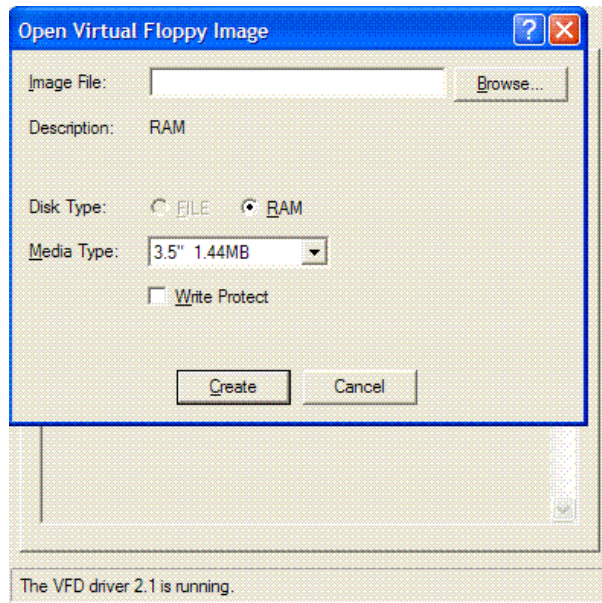
VFD（バーチャル・フロッピー・ドライブ）の使い方

VFDを使って、OSをコピーするための仮想フロッピーイメージを作成します。その使い方を説明します。

1. vfdwin.exeを開きます。
2. ドライバー」タブの「スタート」ボタンをクリックします。これでドライバが起動します。
3. Drive0"、"Drive1"のいずれかのタブをクリックします。
4. 開く」をクリ

ックすると、このよう

に表示されます。



メディアタイプが標準の3.5インチ1.44MBフロッピーで、ディスクタイプがRAMになっていることを確認してください。また、Write Protectが無効になっていることを確認してください。作成」をクリックします。マイコンピュータ」（あなたのコンピュータ）にアクセスすると、新しいフロッピードライブが表示されます。

ディスクをフォーマットするには、ドライブを右クリックして「プロパティ」を開きます。VFDタブの下にフォーマットオプションがあります。

PartCopy - ブートセクターへのコピー

いいですねー。さて、ブートルードの準備ができたところで、それをどうやってディスクにコピーするのでしょうか？ご存知のように、Windowsではディスクの第1セクタに直接コピーすることはできません。このため、コマンドを使ってコピーする必要があります。

2021/11/15 13:03

オペレーティングシステム開発シリーズ

最初のチュートリアルでは、これらのコマンドの1つである「**デバッグ**」*について見てきました。このコマンドを使用することに決めている場合は、このセクションをスキップすることができます。

partcopyです。

PartCopyは、コマンドラインプログラムです。以下のシンタックスを使用します。

```
partcopy ファイル first_byte last_byte ドライブ
```

PartCopyは、単にファイルをコピーするだけではありません。特定のバイトをセクタとの間でコピーするのにも使えます。そのフォーマット（上図）を考えれば、安全な方法です。

あなたはエミュレートされたフロッピードライブを持っているので、ドライブ名をアルファベットで呼ぶこ

とができます（**A:** のように）。私たちのブートローダをコピーするには、次のようにします。

```
partcopy Boot1.bin 0 200 -f0
```

f0 はフロッピーディスク **0** を表します。フロッピーディスクがどのドライブに入っているかによって、f0、f1 など切り替えることができます。Boot1.bin は、コピーするファイルです。これは、ファイルの最初のバイト（0x0）から最後のバイト（0x200、つまり512進数）までをコピーします。partcopyは16進数しか受け付けられないことに注意してください。

警告このプログラムを使用する際には、注意を怠るとパーマネントディスクの破損を引き起こす可能性があることを覚えておいてください。上記のコマンドラインコマンドは、フロッピーディスクに対してのみ動作します。ハードディスクでは絶対に試さないでください。

Bochs: ブートローダのテスト

Bochsは、32ビットのPCエミュレータです。ここでは、Bochsをデバッグやテストに使用します。

Bochsでは、エミュレートするハードウェアを記述したコンフィグレーション・ファイルを使用します。例えば、これは私が使っているコンフィグファイルです。

```
# ROMとVGA BIOSイメージ -----
romimage:    file=BIOS-bochs-latest, address=0xf0000
vgaromimage:VGABIOS-lgpl-latest

# boot from floppy using our disk image -----
floppya: 1_44=a:, status=inserted # ドライブAからの起動

# ログとレポート -----
ログを作成      OSDev.log          # すべてのエラーおよび情報ログは、OSDev.logに出力されます。
します。
のエラーに      action=report
なります。
の情報を提      action=report
供します。
```

設定ファイルでは、コメントに#を使用しています。**A**ドライブに入っているフロッピーディスクイメージ（VFDで作成したものなど）からの起動を試みます。

ROM BIOSとVGA BIOSのイメージはBochsに付属していますので、その心配はありません。

BIOS ROMの配置

設定ファイルのほとんどの行は非常にシンプルです。しかし、ここでは1行だけ見ておく必要があります。

```
romimage:    file=BIOS-bochs-latest, address=0xf0000
```

この行は、Bochsのメモリ(Virtual RAM)のどこにBIOSを配置するかを指示します。BIOSのサイズが異なることを覚えていますか？また、BIOSはメモリの最初のメガバイト(0xFFFFF)の終わりでなければならないことを覚えていますか？

このため、Biosの位置を変更するために、この行を変更する必要があるかもしれません。これは、Biosイメージのサイズを取得することで行うことができます（Bochsディレクトリ内の**BIOS-bochs-latest**という名前になっているはずです）。サイズをバイト単位で取得します。

この後、単純に 0xFFFF - bochs ファイルのサイズ（バイト）を引きます。これが新しいBiosのアドレスになりますので、この行の**アドレス**を更新してBiosを新しい場所に移動させます。

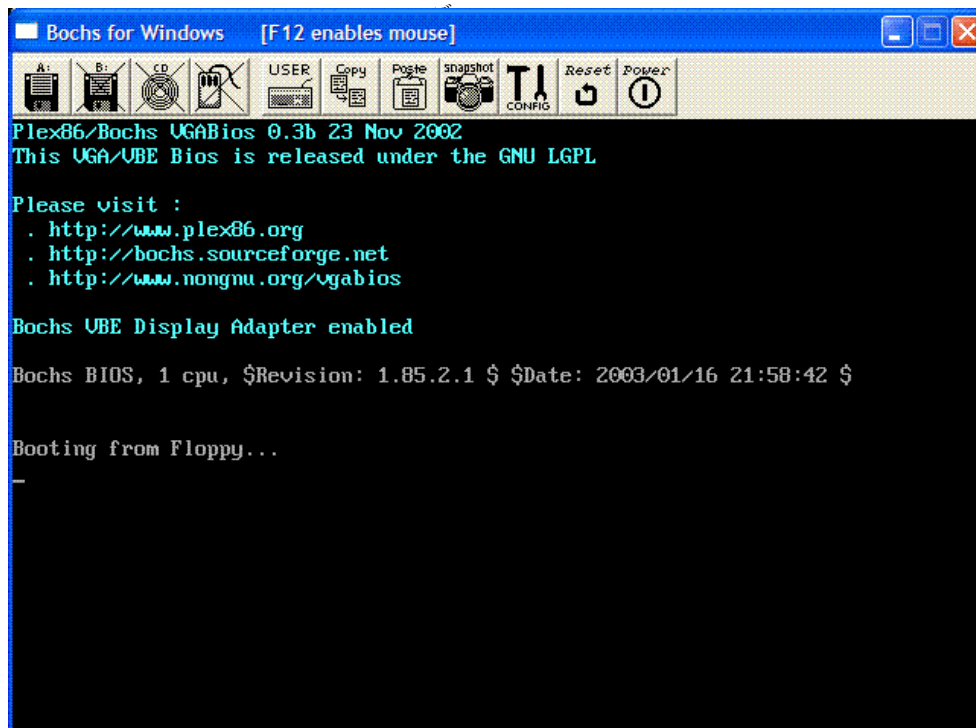
このステップを行う必要がある場合とない場合があります。もし、Biosが0xFFFFで終わらなければならないというエラーがBochsから出た場合は、このステップを完了する必要があり、動作するはずですが。

Bochsの使い方

Bochsを使うために。

1. bochs.exeの実行
2. オプション2（Read options form）を選択し、エンターキーを押します。
3. 設定ファイル名（上記で作成したもの）を入力し、エンターキーを押します。
4. メインメニューに戻ります。オプション5：Begin Simulationを選択し、エンターキーを押

します。新しいウィンドウが開き、次のような画面が表示されます。



もし、ボックスが辞めてしまったり、リスタートしてしまったら

...そうすると、トリプルフォールトが発生したことになります。コードに戻って、どこに問題があるのか探してみてください。何かお困りのことがありましたら、お気軽にご相談ください。

ウィンドウが表示されても何もしない場合

おめでとうございます。これは **cli** と **hlt** 命令がシステムを停止させているので、ブートローダが実行されていることがわかります。

ビルドプロセス - 概要

前回のチュートリアルで説明したビルドプロセスと比較してみてください。慣れてしまえばとても簡単です。ここから先は、ビルドプロセスのこれらのステップを詳しく説明することはありません。

次の機会まで。

マイク

BrokenThorn Entertainment 社。現在、DoE と NeptuneOperating System を開発中です。質問

やコメントはありますか？お気軽にお問い合わせください。

あなたも記事の改善に貢献したいと思いませんか？もしそうなら、ぜひ私に教えてください。



第2章

ホーム

第4





オペレーティングシステム開発シリーズ

オペレーティングシステム開発 - ブートローダー 2

by Mike, 2009

このシリーズは、オペレーティングシステムの開発を一から実演し、教えることを目的としています。

はじめに

Welcome!

前回のチュートリアルでは、多くのことを説明しました。電源ボタンを押したときに何が起るのか、BIOSがどのように起動するのかを見てきました。また、BIOSの割り込み (INT)0x19についても説明しました。この割り込みは、ブートシグニチャ(0xAA55)を検索し、見つかった場合は、0x7C00にあるブートローダをロードして実行します。

また、シンプルなブートローダを開発し、ビルドプロセス全体を経験しました。このチュートリアルでは、

ブートローダについて詳しく説明します。カバーしていきます。

- BIOSパラメタブロックとMBRプロセッ
- サモード
- 割り込み-文字の印刷など セグメント:オフセ
- ットアドレスモード

注: ここから先は、ブートローダがシステム全体をコントロールします。これはつまり、すべてのコードを私たちが書くことに依存するということです。すべては私たち次第なのです。要するに、これからもっとたくさんのコードが出てくるということです。

ここから先は、より複雑な内容になっていきます。このシリーズの構造を確実にものにするために、各チュートリアルにはダウンロード可能なデモを用意しています。これはコンセプトを理解するのに役立ちます。心配しなくても、ここではすべてを詳細に説明します。

いいですか？

プロセッサモード

さてさて、この言葉、どこかで聞いたことがあるような気がします。それは…。すべてのチュートリアルで

それなのに、これまであまり話題になっていませんでした。プロセッサモードの違いを理解することは、私たちにとって非常に重要なことです。これはなぜでしょうか？

前の2つのチュートリアルでは、x86ファミリーが16ビット環境で起動する仕組みと理由について説明しました。今回は、32ビットのオペレーティングシステム (OS) を開発したいので、プロセッサを16ビットモードから32ビットモードに切り替える必要があります。

モードは2つ以上あります。それぞれのモードを見ていきましょう。

リアルモード

ご存知のように、x86プロセッサは16ビット環境で起動します。このモードは何でしょうか？(ヒント: リアルモードではありません) ...

なるほど、そうですね :)とところで、リアルモードのどこがリアルなのでしょう？リアルモード...

- ネイティブの **segment:offset メモリモデル**を使用し
- ています。メモリは1MB (+64k) に制限されています。
- **仮想メモリやメモリ保護機能**はありません。

中にはとても簡単なものもあります。他のものは多少の説明が必要ですが、注意すべき点は、上記のすべてが間接的または直接的にメモリに関係しているということです。

では、もう少し詳しく見てみましょう。

セグメント: オフセット メモリモード: ヒストリー

もう一度過去に戻って、チュートリアル2を見てみましょう。メモリの概念やオペレーティングシステムの使用は、1950年代にさかのぼります。当時のコンピュータはパーソナルコンピュータではなく、大型のメインフレームコンピュータでした。

当時、すべてのコンピュータは非常に大きくてかさばるハードウェアデバイスを持っていたことを思い出してください。時を経て (チュートリアル2を参照)、OSだけでなく、コンピュータも進化していることがわかります。

コンピュータの普及とともに、その需要も高まっていった。コンピュータが8ビットだった頃、多くの人が16ビットを求めていました。16ビットの時代が来たとき、マイクロソフトはすでに32ビットを考えていた。32ビットの時代が来ると同時に、64ビットがすでに主流になっていた。さて、最後の1つは真実ではありませんが :)、128ビットは間もなく登場します。

一番の問題は、コンピューター業界の動きが早いことです。

インテルが8086プロセッサを設計したとき、プロセッサは16ビットのレジスタを使用し、64KBまでのメモリにしかアクセスできなかった。しかし、多くのソフトウェアはそれ以上のメモリを必要としていた。

8086が設計されたのと同じ時期に、8088も設計されていた。しかし、8088はインテルの「次世代」プロセッサになるはずだったが、予想以上に時間がかかっていた。他社に対抗するために、インテルは急遽8086というプロセッサを開発・発売し、8088が発売されるまで我慢しようと考えたのである。

問題は、ソフトウェアが64KB以上のメモリを要求していたことと、インテルのプロセッサ8086は、8088が発売されるまで、すでに16ビットプロセッサを製造していた競合他社に対抗するためのものだったということだ。インテルは、戦略を必要としていた。

8086の設計者は、ある解決策を提案した。この解決策により、8086は16ビットのままで、最大1MBのメモリーにアクセスできるようになる。彼らは賛成し、インテルも承認した。

セグメント: オフセット」という記憶方式が生まれた。

セグメント: オフセットの仕組みを理解するために、まずセグメントとオフセットを分解して見てみましょう。

セグメント

セグメントとは、単純に全体の一部分のことです。ここでは、「セグメント」とはメモリの一部分を指します。そう、基本的にはそ

れだけです。記憶をセクションに分けてみましょう。これらのセクションはセグメントを表します。

2021/11/15 13:04

オペレーティングシステム開発シリーズ

x86ファミリーのプロセッサでは、セグメントの開始位置を格納する4つのプライマリレジスタを使用しています。これはベースアドレスのようなもので、セグメントの開始位置を示します。

通常、1つのセグメントは64KBの大きさで、自由に動かすことができます。

セグメントとは、単にメモリ上の一部分を表すものであることを覚えておいてください。この場合、セグメントのベースアドレスが0であれば、バイト0から64KBまでの区間を表します。

レジスタは**CS、DS、ES、SS**です。これらのレジスターには、セグメントのベースアドレスが格納されています。これらのレジスタはセグメントのベースアドレスを格納するもので、このモードでのアドレッシングを見た後に詳しく見ていきます。

オフセット

オフセットとは、基数に加算される数値のことです。例えば、ベースナンバーが3の場合。

オフセット=ベースナンバー (3) +オフセットナンバー

オフセット2は3+2=5

オフセット4は3+4=7

さて、このことが私たちにどう関係するのでしょうか？セグメント：オフセットアドレッシングでは、ベースアドレス（セグメントはベースアドレスを表すことを覚えておいてください）とオフセットアドレスを加算します。

簡単でしょうか？では、それをまとめてみましょう。

セグメント：オフセットのアドレス指定

セグメント：オフセットのアドレス指定では、セグメントアドレスにオフセットアドレスを加算するだけです。しかし、前節でリアルモードのセグメントアドレスは**16ビット**であると述べました。つまり、セグメントに**16(10進数)**をかけてから、オフセットを加える必要があります。ということで、現在の計算式をご紹介します。

絶対(正確)メモリアドレス = (セグメントアドレス * 16(10進数)) オフセット

それがすべてです :)

セグメント：オフセット規約

セグメントアドレスとオフセットアドレスは、通常、コロン (:) で区切られています。これらのアドレスは、通常、**Segment :オフセット**。例えば、以下のようになります。

07C0:0000< 07C0はセグメント、0はオフセット

上記を計算式で表すと、絶対アドレス0x7C00に変換できます。

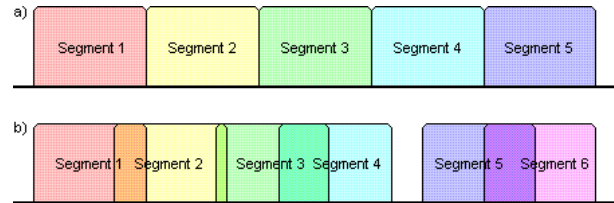
セグメント：オフセット問題 アドレス = ベースアドレス * セグメントサイズ (16) + オフセット 07C0:0000 = 07C0 * 16 (10進数) + 0 = 07C00 + 0 = 0x7C00
セグメント：オフセットは非常にユニークです。セグメントとオフセットの値を変えることで、異なるセグメント：オフセットのペアが同じ絶対アドレスを出力することがわかります。なぜでしょう？それは、どちらも**同じメモリロケーションを参照している**からです。

例えば、以下のアドレスはすべて、0x7C00にあるブートローダを参照しています。

0007:7B90	0008:7B80	0009:7B70	000A:7B60	000B:7B50	000C:7B40
0047:7790	0048:7780	0049:7770	004A:7760	004B:7750	004C:7740
0077:7490	0078:7480	0079:7470	007A:7460	007B:7450	007C:7440
01FF:5C10	0200:5C00	0201:5BF0	0202:5BE0	0203:5BD0	0204:5BC0
07BB:0050	07BC:0040	07BD:0030	07BE:0020	07BF:0010	07C0:0000

これらはほんの一例です。技術的には、メモリ内の同じバイトを参照できる segment:offset の組み合わせは、正確には **4,096** 通りあります。

2つのセグメントアドレスが**64KB以内**にある場合はどうでしょうか。セグメントのサイズ（およびオフセット）は**16ビット**であることを覚えておいてください。また、セグメントアドレスは、セグメントの**ベース**のみを参照しています。これが「**オーバーラップしたセグメント**」です。



レイヤーの上に、他のセグメントの上にレイヤーを重ねることを想像してみてください。これは問題を引き起こす可能性があります。

つまり、リアルモードでは、**4,000**通り以上の方法で、メモリ上のすべてのバイトにアクセスすることができ、知らないうちにメモリの領域を破損する可能性のあるセグメントをオーバーラップさせることができるのです。これが、「リアルモード」が**メモリ保護**を持たないことの意味です。

x86でセグメントの参照に使用されるレジスタは以下の通りです。

- CS (コードセグメント) - コードのベースセグメントアドレスを格納
- DS (データセグメント) - データのベースセグメントアドレスを格納
- ES (エクストラ・セグメント) - 任意のベース・セグメント・アドレスを格納
- SS (スタック・セグメント) - スタックのベース・セグメント・アドレスを格納

リアルモードには多くの問題があります。この問題から私たちを守ってくれるものは何でしょうか？

プロテクトモード

プロテクトモード (PMode) という言葉はよく耳にしますし、これからも耳にすることでしょう。PModeでは、メモリのレイアウトを記述する**ディスクリプターテーブル**を使用して、メモリの保護を行うことができます。

PModeは32ビットのプロセッサモードなので、32ビットのレジスタも使用でき、最大**4GBのRAM**にアクセスできます。Real Modeに比べて大幅に改善されています。

私たちはPModeを使います。そう、WindowsはPMode対応OSなのです。)

PModeは、設定とその仕組みを完全に理解するのが少し難しいです。PModeについては後ほど詳しく説明します。

アンリアル・モード

いつでも好きな時にプロセッサモードを切り替えることが可能です。Unreal Mode」とは、Real ModeをPModeのアドレス空間(4GB制限)で表現した洒落です。

アンリアルモードを有効にするには、プロセッサをリアルモードからPMモードに切り替え、新しいディスクリプターをロードした後、再びPMモードに戻すだけです。配述子テーブルは非常にわかりにくいものです。プロテクトモード(PMode)について詳しく説明するときに、このテーブルについて説明します。

バーチャル8086モード

バーチャル8086モード（v86モード）は、プロテクトモードを16ビットのリアルモードでエミュレートしたモードである。

これはちょっと変だと思いませんか？ しかし、v86は有用です。すべてのBIOS割り込みは、リアルモードでのみ利用可能です！v86モードは、PMode内からBIOS割り込みを実行する方法を提供します。これについては後で詳しく説明します。

プロセッサモードの切り替え

ここでは、プロセッサモードを切り替えるためのコードについては、まだ説明しません。その代わりに、一歩下がって重要なコンセプトを説明したいと思います。

実際のモードとして組み込まれているのは、「Real Mode」と「Protected Mode」の2つだけです。つまり、他のモードである「Unreal Mode」と「v86 Mode」は、この2つのモードから作られているのです。

Unreal ModeはReal Modeですが、Protected Mode (PMode) Addressing systemを使用していることを覚えておいてください。また、バーチャル8086モードはPModeですが、16ビットコードを実行するためにReal Modeを使用しています。

このように、v86モードとUnrealモードは、Real ModeとProtected Modeをベースにしています。このため、PModeを理解していないと、これらのモードの仕組みを理解するのは難しいかもしれません。

PMode、Unreal Mode、v86 Modeについては、近日中に詳しくご紹介しますので、ご安心ください

:)ただし、PModeについてはいくつか重要なことを覚えておきましょう。

- 割り込みは絶対にできません。自分で書く必要があります。ハードウェア、ソフトウェアを問わず、割り込みを使用するとトリプルフォールトが発生します。PMモードに切り替えると、わずかなミスでもトリプルフォールトが発生します。注意してください。
- PModeでは、GDT、LDT、IDTなどの配述子テーブルを使用する必要があります。PModeでは、4GBのメモリにアクセスできます。メモリ保護セグメント付き：オフセットアドレスとリニアアドレスが使用されます。
- 32ビットレジスタのアクセスと使用

PModeについては後ほど詳しく説明します。

ブートローダーの拡張

これまでに多くのことを学んできましたよね？プロテクトモード、アンリアルモード、そしてv86モードの基本的な理論を説明しました。しかし、リアルモードについては深く掘り下げました。なぜかって？DOSとの下位互換性のために、コンピュータは16ビット環境で起動することを覚えておいてください。この16ビットの環境がリアルモードです。

つまり、ブートローダーが実行されると、リアルモードになるのです。ということは、BIOSインターラプトが使えるということですよ？はい。)VGAビデオ割り込みや、ハードウェアから直接マッピングされたその他の割り込みも含まれます:)

使用可能なルーチンとBIOSインターラプト

OEMパラメータブロック

OEMパラメータブロックには、WindowsのMBRとブートレコードの情報が格納されています。このテーブルの主な目的は、ディスク上のファイルシステムを記述することです。このテーブルの説明は、ファイルシステムを見るまでしません。しかし、これがないと先に進めません。

これにより、Windowsの「Not formatted」というメッセージも修正されます。

今のところ、このテーブルは単純に必要なものと考えてください。後でファイルシステムやディスクからのファイルの読み込みについて説明するときに詳しく説明します。このテーブルを使ったブートローダーを示します。

ズ

```

;*****
;      Bootl.asm
;      - シンプルなブートローダ
;
;      オペレーティングシステム開発チュートリアル
;*****

ビット      16
ト
オ          0x7c00
ー
毎開始      jmp ローダ
しまし      ー
だ。
      OEMパラメータブロック
;*****

TIMES 0Bh-$(start) DB 0

      bpbByt

esPerSector:DW 512

      bpbSectorsPerCluster:DB 1
      bpbReservedSectors:DW 1
      bpbNumberOfFATs:DB 2
      bpbRootEntries:DW 224
      bpbTotalSectors:DW 2880
      bpbMedia:DB 0xF0
      bpbSectorsPerFAT:DW 9
      bpbSectorsPerTrack:DW 18
      bpbHeadsPerCylinder:DW 2
      bpbHiddenSectors:DD 0
      bpbTotalSectorsBig:DD 0
      bsDriveNumber:DB 0
      bs未使用:DB 0
      bsExtBootSignature:DB 0x29

```

私たちはまだ16ビットのリアルモードにいます。
0x7C00でBIOSによりロードされます。

OEMブロックを飛び越える

```
bsSerialNumber:      DD 0xa0a1a2a3
bsVolumeLabel:       db "mos floppy"
bsFilesystem:        db
                        "fat"
;*****12M*****;
ブートローダエントリポイント
;*****;

ローダーを使用しています。

        ク      すべてのインターラプトをク
        ラ      リア
        イ      システムを停止させる
回 510 - ($-$) db 0 ; 我々は512バイトでなければならない。残りのバイトを0でクリア
dw 0xAA55
        チ      ブートシグニチャー
        ヤー
        チ
```

文字の印刷 - 割り込み 0x10 関数 0x0E

INT 0x10を使用してビデオ割り込みを行うことができます。ただし、基本的な割り込みしか動作しないことに注意してください。

INT 0x10 - VIDEO TELETYPE OUTPUT

AH = 0x0E
AL = 書くべき文字
BH = ページ番号 (0のはず)
BL = フォアグラウンド・カラー (グラフィックス・モードのみ) 例えば、以下のようになります。

```
        xor     bx, bx      ; BXを0にするためのより速い方法
        movab  ああ, 0x0e
        le     movab      アル, 'A'
        le
        int    0x10
```

これで、文字「A」が画面に表示されます。

文字列の印刷 - 割り込み 0x10 関数 0x0E

同じ割り込みを使って、0終端の文字列を簡単に出力することができます。

```
msg      db      "Welcome to My Operating System!", 0
;*****
; 文字列      を印刷する
;DS=>SI: 0で終了する文字列
;*****

印刷する。
        lodsb
        または      アル、アル
        jz      プリントダウン
                        アル=現在のキャラクター
                        スル・ターミネーターが見つかりました。
                        次の文字を取得する
        movable      ah, 0eh
        int      10h
PrintDoneです。
        jmp     プリント
        レッ
        ト

;*****;
ブートローダエントリポイント
;*****;

loader:
        -----
エラー修正1
        xor     ax, ax      セットアップセグメントが0であることを確認してください。
        movabl  ds, ax      ORG 0x7c00」となります。これは、すべてのアドレスが
        e
        movabl  es, ax      0x7c00:0からです。データセグメントは、同じ
        e      ; コード・セグメント、null em。

        mov     si, msg
        call    Print

        ク      すべてのインターラプトをクリア
        ラ      システムを停止させる
回 510 - ($-$) db 0 ; 我々は512バイトでなければならない。残りのバイトを0でクリア
dw 0xAA55
        チ      ブートシグニチャー
        ヤー
        チ
```

RAMの取得

これは簡単すぎる。

INT 0x12 - BIOS GET MEMORY SIZE

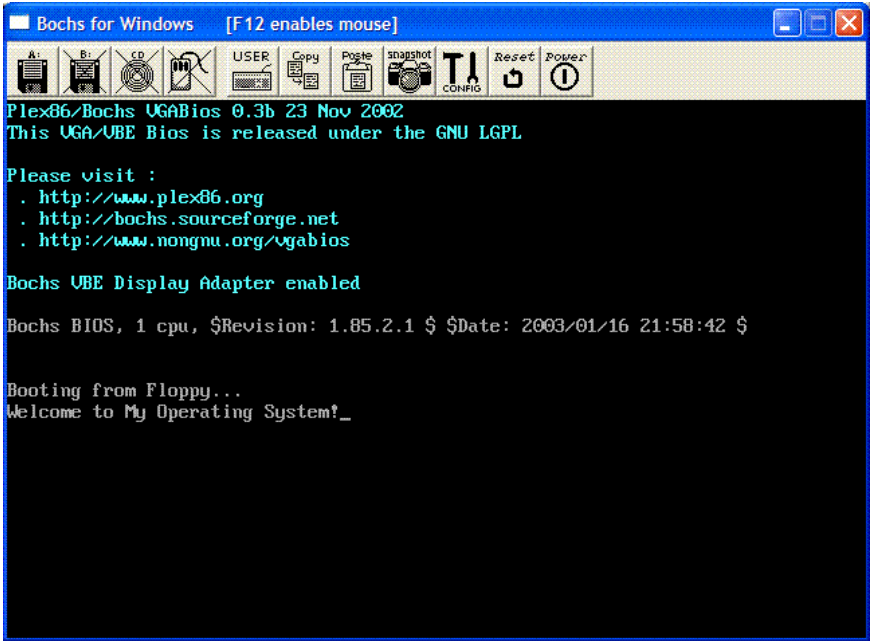
を返します。AX = 絶対アドレス0x0から始まるキロバイト単位の連続したメモ^ズリ。以下に例を示

します。

```
xor     ax, ax
int     0x12
現在 AX=BIOSが記録するシステムのKB量
```

うわぁ…。大変だったでしょう？ :)実は、プロテクトモード (PMode) では、割り込みが使えないので、非常に難しいのです。

注 : BIOSから実際に返されるメモリ量は正確ではないかもしれません。他の方法については後ほどご紹介します。



```
*****
;Boot1.asm
; - シンプルなブートローダ
;
オペレーティングシステム開発チュートリアル
*****

                                bits16; 16ビットのリアルモードのままです。
                                org0x7c00; We are loaded by BIOS at 0x7C00

                                loader; jump over OEM block

start: jmp

*****;
OEMパラメータブロック
*****;

; エラー修正2 - 醜いTIMESディレクティブの削除 -----

;TIMES 0Bh-$$+start DB

                                0; OEMパラメータブロックは正確には3バイトです。
                                積み込んだ場所からこれで、これらの
                                3バイト、さらに8バイト。なぜか？

                                bpbOEMdb "My

                                OS"; このメンバーは正確には8バイトでなければなりません。これは単に
                                OSの名前を入力してください :) その他はすべて同じです。

                                bpbBytesPerSector:DW 512
                                bpbSectorsPerCluster:DB 1
                                bpbReservedSectors:DW 1
                                bpbNumberOfFATs:DB 2
                                bpbRootEntries:DW 224
                                bpbTotalSectors:DW 2880
                                bpbMedia:DB 0xF0
                                bpbSectorsPerFAT:DW 9
                                bpbSectorsPerTrack:DW 18
                                bpbHeadsPerCylinder:DW 2
                                bpbHiddenSectors:DD 0
                                bpbTotalSectorsBig:DD 0
                                bsDriveNumber:DB 0
                                bs未使用:DB 0
                                bsExtBootSign

ature:DB 0x29
                                bsSerialNumbe

r:DD 0xa0a1a2a3
                                bsVolumeLabel:DB "MOS FLOPPY "
                                bsFileSystem:DB "FAT12"

                                msgdb"Welcome to My Operating System!",
                                0; 印刷する文字列

*****
; 文字列 を印刷する
;DS=>SI: 0で終了する文字列
*****

印刷する。

                                lodsb
                                または
                                jz
                                アル、アル
                                プリントダウン

                                movable
                                ah,0e
                                ; Nope-文字を印刷する

                                int
                                h
                                10h
                                jmp
                                プリント
                                ; ヌルのターミネーターが見つかるまで繰り返す

PrintDoneです。
                                レット
                                終わったので、リターン
```

```
;*****;  
ブートローダエントリポイント  
;*****;  
  
loader:
```

<pre>xor ax, ax movab1 ds, ax e movab1 es, ax e movab1 si, msg e コール プリント xor ax, ax int 0x12 ク ラ イ ス ト チ ャ ー チ 回 510 - (\$-\$) db 0 dw 0xAA55</pre>	<p>セットアップセグメントが0であることを確認してください。 ORG 0x7c00」となります。これは、すべてのアドレスが 0x7c00:0からです。データセグメントは、同じ ; コード・セグメント、null em。</p> <p>私たちのメッセージを印刷する 印刷機能呼び出します。</p> <p>クリアアックス BIOSからKBの量を得る</p> <p>すべてのインターラプトをクリア システムを停止させる</p> <p>; 我々は512バイトでなければならない。残りのバイトを0でクリア ブートシグニチャー</p>
---	---

結論

よくぞここまで来たと自分を褒めてあげてください :)

このチュートリアルは、とても難しいものでした。セグメント：オフセットのアドレッシングとプロセッサモードを、あまり深く説明せずに、うまく説明する方法を見つけなければなりませんでした。上手くできたと思います :)

プロセッサのモードには、**Real Mode**、**Protected Mode**、**Unreal Mode**、**v86**があります。ブートローダを開発する際にこのモードを使用することになるので、リアルモードについて詳しく調べました。また、セグメンテーション：オフセットのアドレッシングについても説明しました。これは、**DOS**プログラマーにとっては再教育になるかもしれません。また、いくつかの **BIOS** 割り込みについても調べ、最後に完全な例を示しました。

次のチュートリアルでは、私たちが追加した醜い**OEM**パラメータブロックの解説を行います。また、基本的なファイルシステムの理論と、ディスクからのセクタの読み込みについても見ていきます。

次の機会まで。

マイク
BrokenThorn Entertainment社。現在、DoEとNeptuneOperating Systemを開発中です。 質問やコメントはありますか？お気軽にお問い合わせください。

あなたも記事の改善に貢献したいと思いませんか？もしそうなら、ぜひ私に教えてください。



オペレーティングシステム開発 - ブートローダ 3

by Mike, 2009

このシリーズは、オペレーティングシステムの開発を一から実演し、教えることを目的としています。

ご注意：本章は近日中に更新され、エラーの修正やトピックに関するより詳細な情報の提供が予定されています。

はじめに

Welcome!

前回のチュートリアルでは、さまざまなプロセッサモードと、簡単なBIOSインターラプトについてご紹介しました。また、Real Modでのsegment:offset addressingを見て、Real Modeを詳しく説明しました。また、ブートローダに謎の「OEMパラメータブロック」を追加したり、画面に文字列を表示する機能を追加したりしました。

このチュートリアルでは、アプリケーション・プログラミングとシステム・プログラミングの違いを表すさまざまな「リング」を見ていきます。また、シングルステージとマルチステージのブートローダについて、それぞれの長所と短所を説明します。

最後に、BIOSインタラプト0x13、OEMパラメータブロック、プログラムの読み込み、ロード、実行について説明します。このプログラムがS Stage Bootloaderとなります。Second Stage Bootloaderは、32ビット環境を設定し、C Kernelをロードする準備をします。

いいですか？

アセンブリ言語の輪

アセンブリ言語では、「リング0のプログラム」や「このプログラムはリング3で実行されています」という言葉があります。リングの違いを理解しておくと、OS開発に役立ちます。

リング - 理論

さて、ではリングとは何でしょうか？アセンブリ言語の「リング」は、プログラムがシステムに対して持つ保護と制御のレベルを表しています。リングは4つあります。リング0 1、リング2、リング3です。

リング0のプログラムは、システム内のすべてのものを絶対的に制御することができますが、リング3はそれほど制御できません。リング番号が小さければ小さいほど、そのソフトウェアはより多くの制御権を持っていることになります（レベルプロテクションは少なくなります）。

リングは単なる概念ではなく、プロセッサアーキテクチャーに組み込まれています。

コンピュータが起動したとき、ブートローダが実行されたときも、プロセッサはRing 0にあり、DOSアプリケーションなどほとんどのアプリケーションはRing 3で動作します。つまり、リング0で動作するOSは、通常のリング3のアプリケーションよりもはるかに多くのことを制御できるのです。

スイッチングリング

リングはプロセッサ・アーキテクチャの一部であるため、プロセッサは必要に応じて状態を変化させます。変化するのは...

- ・ファージャンプ、ファークール、ファーターンなど、プログラムを別のリングレベルで実行する指示命令です。トラップ
- ・命令（INT、SYSCALL、SYSENTERなど）。
- ・例外

例外処理については、SYSCALL命令とSYSENTER命令と同様に、後で説明します。

マルチステージブートローダ

シングルステージブートローダ

ブートローダやブートセクタは、512バイトの大きさしかないことを覚えておいてください。その512バイトの中で、ブートローダがカーネルを直接実行した場合、それはSステージブートローダと呼ばれます。

しかし、問題はその大きさにあります。この512バイトの中で何かできることはほとんどありません。16ビットのブートローダで32カーネルをセットアップし、ロードし、実行するのは非常に困難です。これにはエラー処理のコードは含まれていません。これには以下のコードが含まれます。GDT、IDT、A20、PMode、32ビットカーネルのロードと検索、カーネルの実行、エラー処理。これらのコードを512バイトに収めることは不可能です。このため、シングルステージのブートローダでは、16ビットのカーネルをロードして実行しなければなりません。

この問題のため、ほとんどのブートローダはMulti Stage Loaderとなっています。

マルチステージブートローダ

マルチステージブートローダは、512バイトのブートローダ(シングルステージローダ)から構成されていますが、これは別のローダ(セカンドステージブートローダ)をロードして実行するだけです。セカンドステージブートローダは通常16ビットですが、(前のセクションで挙げた)すべてのコードとそれ以上のコードを含みます。32ビットのカーネルをロードして実行できるようになります。

これがうまくいくのは、512バイトの制限がブートローダにしかないからです。ブートローダがセカンドステージ・ローダ用の全セクタを順次ロードする限り、セカンドステージ・ローダのサイズに制限はありません。これにより、カーネルの設定が非常に簡単になります。

今回は、2ステージのブートローダーを使用します。

ディスクからのセクタの読み込み

ブートローダは512バイトに制限されていることを忘れないでください。このため、できることはあまり多くありません。前のセクションで述べたように、私たちはStage Bootloaderを使用する予定です。つまり、Bootloaderはステージ2のプログラム（Kernel Loader）をロードして実行するために必要なのです。

欲を言えば、Stage 2のローダーには、独自の "Choose your Operating System "と "Advanced Options "のメニューを入れることができます :) ああ、あなたの希望は :)

BIOS Interrupt (INT) 0x13 Function 0 - Reset Floppy Disk

BIOSの割り込み0x13は、ディスクアクセスに使用されます。INT 0x13, Function 0を使用して、フロッピードライブをリセットすることができます。これが意味するところは、フロッピーコントローラーがどこから読み込んで、すぐにディスクの最初のセクタに行くということです。

INT 0x13/AH=0x0 - DISK : RESET DISK SYSTEM
AH = 0x0
DL = ドライブ・トゥ・リセット

戻ります。
AH = ステータスコード
CF(Carry Flag)は、成功した場合はクリア、失敗した場合はセットされます。

以下に完全な例を示します。これはフロッピードライブをリセットし、エラーがあれば再試行します。

.リセット ト	あ、0	フロッピーディスク機能のリセット
mov abl e movable int jc	d1、0 0x13 .リセット	ドライブ0はフロッピーディスクドライブ BIOSを呼び出す キャリーフラグ (CF) が設定されている場合は、エラーが発生しています。再度リセット してみてください

なぜこの割り込みが重要なのか？セクターを読み込む前に、セクター0から始まることを確認しなければなりません。フロッピーコントローラがどのセクターから再生しているかはわかりません。これは良くないことで、再起動のたびに変わる可能性があるからです。ディスクをセクタ0に戻すことで、毎回同じセクタを読み取ることができます。

BIOS割り込み(INT)0x13 機能0x02 - セクターの読み込み

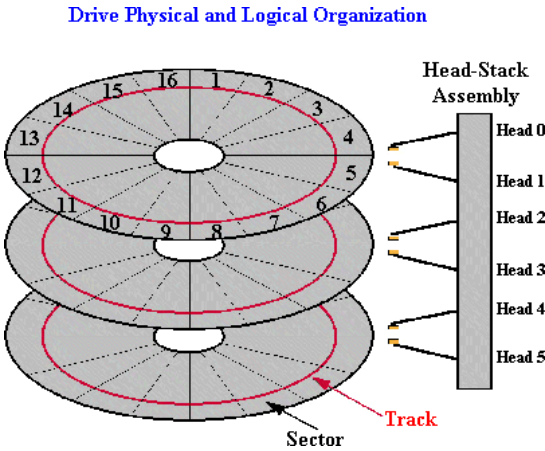
INT 0x13/AH=0x02 - DISK : READ SECTOR(S) INTO MEMORY
AH = 0x02
AL = 読み込むセクタの数
CH = シリンダー番号の下位8ビット
CL = セクター番号 (ビット0〜5)。ビット6-7はハードディスクのみ
DH = ヘッド・ナンバー
DL = ドライブ番号 (ハードディスクの場合はビット7を設定) ES:BX = セクタを読み出すバッファ
ア

戻ります。
AH = ステータスコード
AL = 読み込んだセクタ数
CF = 失敗したらセット、成功したらクリア

さて、これはたくさんのことを考える必要があります。かなり簡単なものもあれば、もっと説明すべきものもあります。では、もう少し詳しく見てみましょう。

CH = シリンダー番号の下位8ビット

シリンダーとは何ですか？円筒とは、円盤上の同じ半径のトラックの集まりのことです。これを理解するために、絵を見てみましょう。



上の写真を見て、思い出してください。

- 各トラックは通常、512バイトのセクタに分割されています。フロッピーでは、1トラックに18セクタあります。
- シリンダーとは、同じ半径を持つトラックの集まりのことです (上の写真の赤いトラックが1つのシリンダーです)。
- フロッピーディスクには2つのヘッドがあります (写真に表示されています)。
- セクター数は2880。

これは何を意味しているのでしょうか？シリンダー番号は、基本的に1枚のディスクのトラック番号を表しています。フロッピーディスクの場合は、転送先のトラックを表しています。

要約すると、1セクタ512バイト、1トラック18セクタ、1サイド80トラックです。

CL = セクター番号 (ビット0〜5)。ビット6〜7はハードディスクのみ

これが最初のセクターで、そこから読み始めます。覚えておってください。1つのトラックには18セクタしかありません。つまり、この値は0から17の間にしかありません。正しいセクタを読むためには、現在のトラック番号を増やし、セクタ番号が正しく設定されていることを確認する必要があります。

この値が18より大きい場合、フロッピーコントローラは、そのセクタが存在しないため、例外を発生させます。ハンドラfoがないため、CPUは2つ目のフォールト例外を発生させ、最終的にはトリプルフォールトになります。

DH = ヘッドナンバー

フロッピーには2つのヘッド (サイド) があることを覚えておってください。ヘッド0は表側で、セクター0があります。そのため、今回は「ヘッド0」を読むことになります。

2021/11/15 13:04

オペレーティングシステム開発シリーズ

この値が2より大きい場合、ヘッドが存在しないため、フロッピーコントローラは例外を生成します。iのハンドラがないため、CPUは2つ目のフォールト例外を生成し、最終的にはトリプルフォールトになります。

DL = ドライブ番号（ハードディスクの場合はビット7を設定） ES:BX = セクタを読み出すバッファ

ドライブ番号とは何ですか？ドライブ番号とは、簡単に言えば、ドライブを表す数字です。ドライブ番号0は通常、フロッピードライブを表します。ドライブ番号1は、5-1/4インチのフロッピードライブを表します。

フロッピーなので、フロッピーディスクから読みたいと思っています。そのため、読み出すためのドライブ番号は「0」です。

ES:BXには、セクタを読み込むためのセグメント：オフセットのベースアドレスが格納されています。ベースアドレスは開始アドレスを表すことを覚えてお

いてください。これを念頭に置いて、セクタを読み込んでみましょう。

セクターの読み出しと読み込み

ディスクからセクターを読み取るには、まずフロッピードライブをリセットしてから読み取る。

.リセット	mov mov int jc	あ、0 dl, 0 0x13 .リセット	フロッピーディスク機能のリセット ドライブ0はフロッピーディスクドライブ BIOSを呼び出す キャリーフラグ (CF) が設定されている場合、エラーが発生しています。再度リセットしてみてください
	mov mov xor	ax, 0x1000 es, ax bx, bx	0x1000:0にセクタを読み込んでみましょう。
.Read	movable mov mov mov mov int jc	ああ、0x02 アル、1 ch、1 cl、2 dh、0 dl、0 0x13 .Read	機能2 1セクタの読み出し 第2セクターを読んでいるので、第1セクターのままです。 読み込むセクタ（第2セクタ 頭部番号 ドライブ番号。ドライブ0はフロッピードライブです。 BIOSを呼び出す - セクタを読み込む Error, so try again
	jmp	0x1000:0x0	セクターを実行するためにジャンプしてください

注：セクタの読み込みに問題があり、そのセクタにジャンプして実行しようとする、セクタが読み込まれたかどうかにかかわらず、CPUはそのアドレスの命令を実行します。これは通常、CPUが無効な命令やunkown命令、またはメモリの終端に遭遇することを意味し、いずれもトリプルフォールトとなります。

上記のコードは生のセクタを読み込んで実行するだけなので、私たちのニーズにはちょっと無意味です。現在、PartCopyは512bだけをコピーするように設定されています。を意味する。どこで、どのようにして、生のセクターを作るのか。

また、この生のセクターに「ファイル名」をつけることは不可能です。ただの生のセクターなのだ。

最後に、現在ブートローダはFAT12ファイルシステム用に設定されています。Windowsはセクター2と3から特定のテーブル(File Allocation Tables)を読み取ろうとしますが、Rawセクターではこれらのテーブルは存在しないため、Windowsは(テーブルであるかのように)ゴミのような値を取ります。その結果は？Windowsでフロッピーディスクを読み込むと、ファイルやディレクトリの名前がおかしくなったり、サイズが大きくなったりします（3.14MBのフロッピーディスクに2.5GBのファイルが入っているのを見たことがありますか？私はあります：）。

もちろん、この方法でセクターをロードする必要があります。しかし、その前に、適切にロードするためには、開始セクタ、セクタ数、ベースアドレスなどを見つけなければならない。これがディスクからのファイル読み込みの基本である。

次はこれを見てみましょう。

FAT12ファイルシステムの操作

OEMパラメータブロック - 詳細

前回の記事では、コードの中に醜いテーブルを捨てました。これは何だったのでしょうか？ああ、そうか...

bpbBytesPerSectorです。	DW	512
bpbSectorsPerClusterです。	DB	1
bpbReservedSectorsです	DW	1
bpbNumberOfFATs:	DB	2
bpbRootEntriesです。	DW	224
bpbTotalSectorsです。	DW	2880
bpbMedia:	DB	0xF0
bpbSectorsPerFATです。	DW	9
bpbSectorsPerTrackです	DW	18
bpbHeadsPerCylinder:	DW	2
bpbHiddenSectorsです。	DD	0
bpbTotalSectorsBig:	DD	0
bsDriveNumberです。	DB	0
bsUnusedです。	DB	0
bsExtBootSignatureです	DB	0x29
bsSerialNumberです。	DD	0xa0a1a2a3
bsVolumeLabel:	db	"mos floppy"
bsFileSystemです。	DB	"FAT12"

これらの多くはとてもシンプルです。これを詳しく分析してみましょう。

bpbBytesPerSector:DW	512
bpbSectorsPerCluster:DB	1

bpbBytesPerSectorは、1セクタを表すバイト数を示します。通常、フロッピーディスクの場合は512バイトです。 bpbSectorsPerClusterは、クラスタあたり

のセクタ数を示します。ここでは、1クラスタあたり1セクタとしています。

```
bpbReservedSectors:    DW 1
bpbNumberOfFATs:       DB 2
```

ズ

予約セクターとは、FAT12に含まれていないセクターの数です。ここでは、ブート^ブロードを格納する**ブートセクタ**は、このディレクトリに含まれません。このため、**bpbReservedSectors**は1とします。

これは、予約されたセクタ(Our bootloader)には、File Allocation Tableが含まれないことを意味します。

bpbNumberOfFATs rは、ディスク上の**FAT (File Allocation Tables)** の数を表す。**FAT12ファイル・システムでは、常に2つのFATがあります。**

通常は、これらのFATテーブルを作成する必要があります。しかし、VFDを使用しているので、**Windows/VFDがディスクをフォーマットするときに、これらのテーブルを作成するようにすることができます。**

注意：これらのテーブルは、エントリを追加または削除したときに、**Windows/VFDによって書き込まれます。つまり、新しいファイルやディレクトリを追加したときに書き込まれます。**

```
bpbRootEntries:      DW 224
bpbTotalSectors:     DW 2880
```

フロッピーディスクの**ルートディレクトリ**内には、最大224個のディレクトリがあります。また、**フロッピーディスクには2,880個のセクタがあることを覚えておいてください。**

```
bpbMedia:            DB 0xF0
bpbSectorsPerFAT:    DW 9
```

メディアディスクリプターバイト (bpbMedia) は、ディスクに関する情報を格納したバイトコードです。このバイトは、ビットパターンです。

- **ビット0 : Sides/Heads** = 片面の場合は0、両面の場合は1
- **ビット1 : サイズ**=FATあたり9セクタあれば0、8セクタあれば1。
- **ビット2 : 密度**=80トラックの場合は0、40トラックの場合は1。
- **ビット3 : タイプ** = 固定ディスク（ハードドライブなど）の場合は0、リムーバブルディスク（フロッピードライブなど）の場合は1。
- **ビット4〜7**は未使用で、常に1です。

0xF0=1110000のバイナリ。これは、**片面、FATあたり9セクタ、80トラック、ムーバブルディスクであることを意味します。bpbSectorsPerFATを見てみると、やはり9です。**

```
bpbSectorsPerTrack:  DW 18
bpbHeadsPerCylinder: DW 2
```

bpbHeadsPerCylinderは、単純に1つのシリンダーを表す2つのヘッドがあることを表しています。シリンダーが何であるかわからない場合は、セクション「**BIOS Interrupt (INT) 0x13**」の「**Read Sectors**」をお読みください。)

```
bpbHiddenSectors:DD 0
```

物理ディスクの先頭からボリュームの先頭までのセクタ数を表します。

```
bpbTotalSectorsBig:  DD 0
bsDriveNumber:       DB 0
```

フロッピードライブがドライブ**0**であることを覚えていますか？

```
bsUnused:            DB 0
bsExtBootSignature:   DB 0x29
```

Boot Signatureは、この**BIOSパラメータブロック**（このOEMテーブル）の種類とバージョンを表します。値は次のとおりです。

- 0x28および0x29は、MS/PC-DOSバージョン4.0のBios Parameter Block（BPB）であることを示します。

0x29があるので、これが使用しているバージョンです。

```
bsSerialNumber:      DD 0xa0a1a2a3
bsVolumeLabel:       db "mos floppy"
bsFileSystem:        db
                     "fat12"
```

シリアルナンバーは、フォーマットを行うユーティリティーによって割り当てられます。シリアルナンバーは、そのフロッピーディスクに固有のものであり、2つのシリアルナンバーが同一になることはありません。 Microsoft、PC、DR-DOSでは、シリアルナンバーは以下のように現在の日時をベースにしています。

```
低位16ビット = ((秒 + 月) << 8) + (100分の1 + day_of_month) 高位16ビット
= (時間 << 8) + 分 + 年
```

シリアルナンバーは上書きされるので、何を入れても問題ありません。

Volume Labelは、ディスクに何が入っているかを示す文字列です。OSによってはこれを名前として表示するものもあります。**注：この文字列は11バイトでなければなりません（※）。それ以上でもそれ以下でもありません。**

Filesystem文字列は、同じ目的で使用されますが、それ以上ではありません。**注：この文字列は8バイトでなければならない、それ以上でもそれ以下でもありません。**

デモ

わあ、すごい量ですね。以下は、このチュートリアルで開発したブートローダで、すべてをまとめています。

注意：この**デモはそのままでは動作しません**。このデモは、もともとデモンストレーションを目的としたものであり、現在の状態ではビルドできません。このチュートリアルを更新し、デモをビルドできるようにする予定です。

```
*****
;      Bootl.asm
;      - シンプルなブートローダ
;
;      オペレーティングシステム開発チュートリアル
; *****
```

ビット 16
ト

私たちはまだ16ビットのリアルモードにいます。

0x7C00でBIOSによりロードされます。

オーガ	0x7c00	
を開始しました。	jmp ローダー	OEMブロックを飛び越える
;*****; OEMパラメータブロック / BIOSパラメータブロック ;*****;		
TIMES 0Bh-\$+start DB 0		
	bpbByt	
esPerSector:DW 512		
	bpbSectorsPerCluster:DB 1 bpbReservedSectors:DW 1 bpbNumberOfFATs:DB 2 bpbRootEntries :DW 224 bpbTotalSectors:DW 2880 bpbMedia:DB 0xF0 bpbSectorsPerFAT:DW 9 bpbSectorsPerTrack:DW 18 bpbHeadsPerCylinder:DW 2 bpbHiddenSectors:DD 0 bpbTotalSectorsBig:DD 0 bsDriveNumber:DB 0 bs未使用 : DB 0 bsExtBootSign	
ature:DB 0x29	bsSerialNumbe	
r:DD 0xa0a1a2a3	bsVolumeLabel:DB "MOS FLOPPY " bsFileSystem:DB "FAT12"	
;*****; ;文字列を印刷する ;DS=>SI: 0で終了する文字列 ;*****; 印刷する。		
	lodsb or al, al jz PrintDone mov ah, 0eh int 10h jmp プリント	SIからALへの文字列から次のバイトを読み込む AL=0の場合は? そうだ、ヌルターミネーターが見つかったから脱出しよう ; Nope=文字を印刷する
PrintDoneです		;ヌルのターミネーターが見つかるまで繰り返す
	レット	終わったので、リターン
;*****; ブートルードエントリポイント ;*****;		
ローダー を使用し ています 。		
.リセッ ト	movab あ, 0 le dl, 0 mov int 0x13 jc .リセット	フロッピーディスク機能のリセット ドライブ0はフロッピーディスクドライブ BIOSを呼び出す キャリフラグ (CF) が設定されている場合は、エラーが発生しています。再度リセット してみてください
	mov ax, 0x1000 mov es, ax xor bx, bx	0x1000:0にセクタを読み込んでみましょう。
	mov ah, 0x02 mov al, 1 mov ch, 1 mov cl, 2 int dh, 0 int dl, 0 0x13	フロッピーディスクのセクター読み出し機能 1セクタの読み出し 第2セクタを読んでいるので、第1セクタのままです。 読み込むセクタ (第2セクタ 頭部番号 ドライブ番号。ドライブ0はフロッピードライブです。 BIOSを呼び出す - セクタを読み込む
	jmp 0x1000:0x0	セクタを実行するためにジャンプしてください
ア		times 510 - (\$-\$) db 0; 我々は512バイトでなければなら残りのバイトを0でクリ
		dw 0xAA55; Boot Signature
; セクタ1の終了、セクタ2の開始 -----		
		org 0x1000; このセクタはブートセクタによって0x1000:0にロードされる
		clic; システム
halt		

結論

ディスクの読み込みやBIOS Parameter Block (BPB) についても詳しく説明しました。また、これらを組み合わせた簡単なデモも作成しました。

また、アセンブリ言語のリングの違いを見てみると、OSはリング0で、他のプログラムとは異なることがわかりました。これにより、アプリケーションプログラムにはない特別な特権的命令を使用することができます。

これで、セカンドステージのローダを見つけてロードするために必要なものがすべて揃いました!次のチュートリアルでは、FAT12についてのすべてを学び、第2ステージをロードします。お楽しみに:)次の機会まで。

2021/11/15 13:04

オペレーティングシステム開発シリーズ

マイク

ズ

BrokenThorn Entertainment社。現在、DoEと[Neptune Operating System](#)を開発中です。質問

やコメントはありますか？お気軽に[お問い合わせ](#)ください。

あなたも記事の改善に貢献したいと思いませんか？もしそうなら、ぜひ[私に教えて](#)ください。





オペレーティングシステム開発シリーズ

オペレーティングシステム開発 - ブートローダ 4

by Mike 2009

このシリーズは、オペレーティングシステムの開発を一から実演し、教えることを目的としています。

はじめに

ようこそ!前回のチュートリアルでは、セクタをロードして実行する方法についてお話ししました。また、アセンブリ言語でのRingsの説明や、BIOS Parameter Block (BPB) の詳細についても見てきました。

このチュートリアルでは、FAT12ファイルシステムを解析し、2番目のステージローダーを名前でロードするために、学んだことをすべて使用します。

このチュートリアルには、たくさんのコードが含まれています。私はすべてを詳細に説明するように最善を尽くします。また、このチュートリアルには、いくつかの数学も含まれています。

いいですか？

クリとHLT

なぜ私がデモプログラムの最後に "cli" と "hlt" という指示を出すのか、不思議に思われるかもしれません。それはとても簡単なことです。何らかの方法でプログラムを停止させないと、CPUはプログラムを超えてランダムな命令を実行してしまうのです。そうすると、最終的にはトリプルフォールトになってしまいます。

割り込み(cli)をクリアするのは、割り込みを実行したくても実行されてしまう(つまりシステムが停止しない)からです。これは問題を引き起こす可能性があります。つまり、(cliを使わずに) hlt命令を実行するだけで、cpuがトリプルフォールトになってしまうのです。

そのため、私はいつもデモの最後にcliとhltを使うようにしています。

ファイルシステム - 理論

Yippe!ファイルシステムについてお話ししましょう。)

ファイルシステムは、単なる仕様です。ディスク上に「ファイル」という概念を作り出すためのものです。

ファイルとは、何かを表現したデータの集まりです。このデータは、私たちが望むものであれば何でもあります。データをどのように解釈するかにかかっています。

ご存知のように、1セクタは512バイトの大きさです。ファイルはこのセクタを越えてディスクに格納される。ファイルが512バイトを超える場合は、より多くのセクタを与えなければなりません。すべてのファイルが均等に512バイトのサイズではないので、残りのバイト（ファイルが使用しないバイト）を埋める必要があります。ブートローダでやったことと同じですね。

ファイルが複数のセクタにまたがっている場合、FATファイルシステムではこれらのセクタをクラスタと呼びます。例えば、カーネルは多くのセクタにまたがっている可能性があります。カーネルをロードするには、カーネルが配置されているクラスタ（セクタ）をロードする必要があります。

ファイルが異なるクラスタの異なるセクタにまたがっている場合（Not contiguous）、そのファイルはフラグメンテーションされているといえます。ファイルの異なる部分を集める必要があります。

ファイルシステムには様々な種類があります。広く使われているもの（FAT12、FAT16、FAT32、NTFS、ext（Linux）、HFS（古いMACで使用）など）もあれば、特定の企業が社内でのみ使用するファイルシステム（GFS - Google File Systemなど）もあります。

多くのOS開発者は、FATファイルシステムのバージョン（あるいは全く新しいもの）を作成しています。これらは通常、最も一般的なファイルシステム（FATやNTFSなど）には及びません。

さて、これでファイルシステムについて少しは理解できたと思います。ここでは、シンプルなFAT12を使用します。決めたら、いつでも別のものを使うことができますよ：)

FAT12ファイルシステム - 理論

FAT12は、1977年に発表された最初のFAT（File Allocation Table）ファイルシステムで、Microsoft Disk BASICで使用された。FAT12は古いファイルシステムであり、一般的にfloppyディスク用としてリリースされたため、いくつかの制限があった。

- FAT12は階層的なディレクトリをサポートしていません。つまり、**The Root Directory**という1つのディレクトリしかありません。クラスタ・アドレスは12ビットしかなく、クラスタの最大数は4096に制限されていました。
- **ファイル名は12ビットの識別子としてFATに格納されます。クラスタ・アドレスは、ファイルの開始クラスタを表します。**
- ディスク・サイズは16ビットのセクタのカウントとしてのみ保存され、32MiBのサイズに制限されています
- **れています** FAT12はパーティションを識別するために "0x01" を使用します
-

これらは大きな制限です。では、なぜFAT12が必要なのか？

FAT16は、FAT12に比べて16ビットのクラスタ(ファイル)アドレスを使用するため、ディレクトリや64,000以上のファイルをサポートしています。しかし、**FAT16とFAT12は非常に似ています。**

シンプルにするために、ここではFAT12を使用します。後でFAT16（あるいはFAT32）に変更するかもしれません。(FAT32はFAT12/16とはかなり違うので、後でFAT16を使うことになるかもしれません。)

FAT12ファイルシステム - ディスクストレージ



ブートセクタ	エクストラリザーブドセクション	ファイルアロケーションテーブル1	ファイルアロケーションテーブル2	ルートディレクトリ (FAT12/FAT16のみ)	ファイルやディレクトリを含むデータ領域
--------	-----------------	------------------	------------------	---------------------------	---------------------

これは典型的なフォーマットされたFAT12ディスクで、ブートセクタからディスクの最後のセクタまでを示しています。この構造を理解することは、ファイルを読み込んだり検索したりする際に重要になります。

1つのディスクには2つのFATがあることに注意してください。ルートディレクトリは予約済みのセクタ(予約がない場合はブートルード)の直後にあります。また、Root DirectoryはすべてのFATの直後にあることに注意してください。これはつまり...

FATごとのセクタ数と予約セクタを足すと、最初のセクタをルートディレクトリにすることができます。ルートディレクトリで単純な文字列(ファイル名)を検索すれば、ディスク上のファイルの正確なセクタを見つけることができます。)

もっとよく見てみると...

ブートセクター

このセクションには、BIOSパラメータブロックとブートルードが含まれています。ああ、そうだ。BIOS Parameter Blockには、私たちのディスクを説明するための情報が含まれています。

エクストラリザーブドセクション

BPBのbpbReservedSectorsメンバーを覚えていますか？余分な予約セクタは、ブートセクタのすぐ後に、ここに格納されます。

ファイルアロケーションテーブル(FAT)

クラスタはディスク上の一連の連続したセクタを表すことを覚えておいてください。各クラスタのサイズは通常2KBから32KiBです。ファイルピースは、リンクリストなどの一般的なデータ構造を用いて、あるクラスタから別のクラスタへとリンクされます。

FATは2つあります。しかし、1つはデータ復旧のために1つ目をコピーしただけです。実際には使われていません。

ファイルアロケーションテーブル (FAT) は、これらのクラスタのそれぞれに対応するエントリのリストです。これらのクラスターにデータを保存する際に役立つ重要な情報を特定するのに役立ちます。

各エントリは、クラスタを表す12ビットの値です。FATはこれらのエントリでリンクされたリストのような構造になっており、どのクラスタが使われているかを識別するのに役立ちます。

これを理解するために、可能な値を見てみましょう。

- 値マークの空きクラスタ : 0x00
- Value marks Reserved cluster : 0x01
- このクラスターは使用中であり、次のクラスターを示す値は0x002から0xFFEである。
- 予約値 : 0xFF0~0xFF6
- パッドクラスターの値 : 0xFF7
- 値は、このクラスタがファイルの最後であることを示す。

FATはこれらの値を配列したものに過ぎません。ルートディレクトリから開始セクタを見つけると、FATを調べてどのクラスタをロードするかを見つけることができます。どうやって？単純に値をチェックします。値が0x02から0xFFEの間であれば、この値はファイルをロードするための次のクラスタを表します。

これをもう少し詳しく見てみましょう。ご存知のように、クラスターは一連のセクタを表しています。そのセクタ数は、BIOS Parameter Blockから定義します。

```
bpbBytesPerSector:DW 512
bpbSectorsPerCluster:DB 1
```

今回のケースでは、各クラスタは1セクタです。ステージ2の最初のセクタを取得すると(ルートディレクトリから取得します)、このセクタをFATの開始クラス番号として使用します。開始クラスタを見つけたら、FATを参照してクラスタを決定します (FATは単なる32ビットの数字の配列です。この番号を上リストと比較して、それをどうするかを決定するだけです。)

ルートディレクトリテーブル

今は、これが私たちにとって重要なことです。)

ルートディレクトリは、ファイルやディレクトリに関する情報を表す32バイトの値のテーブルです。この32バイトの値は、フォーマットを使用しています。

- バイト 0-7 :DOSファイル名 (スペースでパッドされている) 8
- ~10バイト目DOSファイルの拡張子 (スペースでパッドされてい
- る) バイト11 :ファイルの属性。これはビットパターンです。
 - ビット0 :Read
 - Only Bit 1 :
 - Hidden Bit 2 : System
 - Bit 3 : ボリュームラベル
 - ビット4 : これはサブディレクトリです
 - Bit 5 : アーカイブ
 - ビット6 : デバイス (内部使用
 - ビット6 : 未使用
- Bytes 12 :未使用
- Bytes 13 :作成時間 (単位 : ms
- 14~15バイト目 : 作成時刻 (以下のフォーマットで表示
 - ビット0-4 : 秒数 (0-29

2021/11/15 13:05

オペレーティングシステム開発シリーズ

- ビット **5-10** : 分 (0~59)
- Bit **11-15** : 時間(0-23)
- **16byte-17byte** : 作成した年を以下の形式で表示します。
 - ビット**0-4** : 年 (0=1980、127=2107)
 - ビット**5-8** : 月 (1=1月、12=12月)
 - ビット**9-15** : 時間(0-23)

- **Bytes 18-19 :Bytes 20-21 :** EAインデックス (OS/2やNTで使用され
- ていますが、気にしないでください) **Bytes 22-23 : Last Modified**
- **time** (フォーマットはバイト**14-15**を参照してください。 **Bytes 22-23**
- **: Last Modified time** (フォーマットはbyte 14-15参照) **Bytes 24-25 :**
- Last modified date (フォーマットはbyte 16-17参照) **Bytes 26-27 :**
- **First Cluster** (クラスター)ファーストクラスター
- **Bytes 28-32 :** ファイルサイズ

重要な部分は太字にしましたが、それ以外の部分はMicrosoftが追加した単なるゴミで、ずっと後にFAT12ドライバを作成するときに追加することになります。ちょっと待ってください。DOSのファイル名は11バイトに制限されているのを覚えていますか？これがその理由です。

- バイト **0-7 :DOSファイル名** (スペースでパッドされている) **8**
- **~10**バイト目**DOSファイルの拡張子** (スペースでパッドされている)

0~10は、うーん、11バイトですね。ファイル名が11バイトに満たないと、データ入力の際にミスが生じます(上に表示されている32バイトの入力テーブル)。もちろん、これは悪いことです。)このため、ファイル名を文字で埋めて、11バイトであることを保証しなければなりません。

前回のチュートリアルで、ファイル名には**内部ファイル名**と**外部ファイル名**があることを説明しましたが、覚えていますか？今回説明したファイル名の構造は、内部ファイル名です。11バイトに制限されているので、**"Stage2.sys "**というファイル名は、次のようになります。

"STAGE2 SYS" (パディングに注意！)

FAT12を検索して読む - 理論編

さて、上記のすべてを読んだ後、あなたは私が「FAT12」と言うことに飽きたでしょう :)そ

れはさておき...。彼の情報は私たちにとってどのように役に立つのでしょうか？

今回は、BIOS Parameter Block (BPB)について説明します。ここでは、以前のチュートリアルで作成したBPBを参照します。

```
bpbBytesPerSectorです。    DW 512
bpbSectorsPerClusterで    DB 1
す。
bpbReservedSectorsです    DW 1
。
bpbNumberOfFATs:         DB 2
bpbRootEntriesです。      DW 224
bpbTotalSectorsです。     DW 2880
bpbMedia:                 DB 0xF0
bpbSectorsPerFATです。     DW 9
bpbSectorsPerTrackです    DW 18
。
bpbHeadsPerCylinder:      DW 2
bpbHiddenSectorsです。     DD 0
bpbTotalSectorsBig:       DD 0
bsDriveNumberです。        DB 0
bsUnusedです。             DB 0
bsExtBootSignatureです    DB 0x29
。
bsSerialNumberです。       DD 0xa0a1a2a3
bsVolumeLabel:            DB " MOS FLOPPY
                             "
bsFileSystemです。         DB "FAT
                             12"
```

各メンバーについての詳しい説明は、前回のチュートリアルをご覧ください。

今回の目的は、2段目のローダーをロードすることです。では、具体的にどのような作業が必要なのかを見ていきましょう。

ファイル名で始まる

まず最初にすべきことは、良いファイル名を作ることです。覚えておいてください。**ルートディレクトリを破壊しないように、ファイル名は正確に11バイトでなければなりません。**

私は2段目に「STAGE2.SYS」を使用しています。その内部ファイル名の例を上記のセクションで見ることができます。

ステージ2の作成

さて、Stage2はブートローダとは別のプログラムです。このStage2はDOSのCOMプログラムに非常に似ています。今、Stage2がしている

ことは、メッセージを表示して停止するだけです。ブートローダですで見ただけのものばかりです。

註：ここでは、普通に実行されています。
COMプログラムでは、まだRing 0の状態です。
; このローダを使って、32ビットの
; モードと基本的な例外処理

; このロードされたプログラムが32ビットのカーネルになります
。

ここでは、512バイトという制限はありません。
なにか0x0ここに何かを加えることで、この後、セグメントを設定する。

ビット16 私たちはまだ現実のモードにいます
。
リニアアドレス0x10000にロードされます。

jmp メイ 本文へジャンプ
ン
; *****;
; ; 文字列 を印刷する

```
; DS=>SI: 0で終了した文字列
;*****;

印刷する。                                lodsb          ; SIからALまでの文字列から次のバイトをロード al,
                                          または al ; Does AL=0?
のメインとなります。                     jz            プリントダ そうだ、ヌルターミネーターが見つかったから脱
す。                                     mov edi, edi 割り込みの解除よう
                                          mov ecx, 10H ; Nope-文字を印刷する
PrintDone                                mov ecx, 99H   ; No-文字が見つかるまで繰り返
です。                                  shl          ds
                                          ; シフト
                                          ; ポイント
                                          ; シフト
;*****;
セカンドステージロードのエンターボイスil; Msg
;*****;

                                          ク          トリプルフォールトを防ぐためのクリアな割込み
                                          ラ          システムの停止
                                          イ
                                          ス
                                          ト
                                          チ
                                          ヤ
                                          ー
                                          チ

;*****;
データセクション
;*****;

Msgdb"Preparing to load operating system...",13,10,0
```

NASMでアセンブルするには、バイナリプログラム（COMプログラムはバイナリ）としてアセンブルして、フロッピーディスクイメージにコピーすればよい。例えば、以下のようになります。

```
nasm -f bin Stage2.asm -o STAGE2.SYS

copy STAGE2.SYS A:\STAGE2.SYS
```

PARTCOPYは必要ありません :)

Step 1: ルートディレクトリテーブルの読み込み

さて、いよいよStage2.sysをロードします。ここでは、Rootディレクトリテーブルと、ディスク情報のためのBIOSパラメータブロックを参照しています。

ステップ1：ルートディレクトリのサイズの取得

さて、まずはルートディレクトリのサイズを取得する必要があります。

サイズを知るには、ルートディレクトリにあるエントリの数を掛けるだけです。簡単そうですね :)

Windowsでは、FAT12フォーマットのディスクにファイルやディレクトリを追加すると、Windowsが自動的にファイル情報をルートディレクトリに追加するので、気にする必要はありません。これにより、作業が非常にシンプルになりました。

ルート・エントリの数をセクタあたりのバイト数で割ると、ルート・エントリが使用しているセクタ数がわかります。

ここではその一例をご紹介します。

```
mov     ax, 0x0020          32バイトのディレクトリエント
mul     WORD [bpbRootEntries] ; ルートエントリの数
div     WORD [bpbBytesPerSector] ; ルートディレクトリで使用されているセクタ
                                の取得
```

ルートディレクトリテーブルは、ファイル情報を表す32バイトの値（エントリ）のテーブルであることを覚えておってください。

よし、これでルートディレクトリにロードするセクターの数がわかった。では、ロードするための開始セクターを見つけましょう :)

ステップ2：ルートディレクトリの起動

これも簡単です。まず、FAT12フォーマットのディスクをもう一度見てみましょう。

ブートセクタ	エクストラリザードセクション	ファイルアロケーションテーブル	ファイルアロケーションテーブル	ルートディレクトリ (FAT12/FAT16のみ)	ファイルやディレクトリを含むデータ領域
--------	----------------	-----------------	-----------------	---------------------------	---------------------

ー	ン	ル1	ル2	ズ		
---	---	----	----	---	--	--

さて、ルートディレクトリは、**FATと予約済みセクタの両方の直後に位置している**ことに注意してください。つまり、**FATs+予約セクタ**を足せば、ルートディレクトリが見つかるということです。

例えば・・・。

```
mov    al, [bpbNumberOfFATs] ; FATの数を取得 (使用頻度2)
mul    [bpbSectorsPerFAT] ; FATの数 * FATあたりのセクタ数; セクタ数の取得 ax,
add    [bpbReservedSectors] ; 予約セクタの追加
```

ここで、AXはルートディレクトリの開始セクタです。

とても簡単でしょう？あとは、セクタをメモリのどこかの位置に読み出すだけです。

ルートディレクトリ - 完全な例、0x0200 ; ルートディレクトリを7c00:0x0200にロード
ード callReadSectors
このサンプルコードは、チュートリアル最後の出てくるブートローダから直接引用しています。ルートディレクトリをロードします。

```
LOAD_ROOTです。

ルートディレクトリのサイズを計算し、"cx "に格納する。

xor    cx, cx
xor    dx, dx
movab  ax, 0x0020          32バイトのディレクトリエントリ
le     マルチ WORD [bpbRootEntries]   ディレクトリの合計サイズ
div    WORD [bpbBytesPerSector]使用  ディレクトリが使用するセクタ
      用します。
xchg   ax, cx
ルートディレクトリの位置を計算し、"ax "に格納する。

mov     al, BYTE [bpbNumberOfFATs]; FATの
mul     数
add     WORD [bpbSectorsPerFAT]; FAT
mov     で使用されるセクタ ax, WORD [bpbReservedSectors]; ブ
add     ートセクタの調整 WORD [datasector], ax; ルートディレク
      トリのベース WORD [datasector], cx
ルートディレクトリをメモリに読み込む(7c00:0200)

mov     bx, 0x0200          ブートコードの上にルートディレクトリ
call    ReadSectors         をコピー
```

ステップ2：ステージ2を探す

さて、これでルートディレクトリのテーブルが読み込まれました。上のコードを見ると、**0x200に読み込まれています**。さて、ファイルを探すためにもう一度、32バイトのルートディレクトリテーブルを見てみましょう（セクション「ルートディレクトリテーブル」。最初の11バイトはファイル名を表しています。また、ルートディレクトリの各エントリは32バイトなので、32バイトごとに次のエントリの開始点となり、次のエントリの最初の11バイトに戻ることになります。

したがって、ファイル名を比較し、次のエントリ（32バイト）にジャンプし、セクタの最後に到達するまで再度テストするだけでよいのです。例えば...

```
バイナリイメージのルートディレクトリの参照
mov     cx, エントリの数を調べます。0になるとファイルが存在しない
mov     [bpbRootEntries], ルートディレクトリがここに読み込まれま
.LOOP:   ず。 0x0200 した。
      ユ
      す。 movab  cx, 11          11文字の名前
      e
      movabl si, ImageName   この11バイトをファイル名と比較します。
      e
      プッシ ディ
      ユ
      リポート エントリーマッチのテスト
      CMPSB
      ポップ ディ
      ジ LOAD_FAT          が一致したので、FATの読み込みを開始します。
      ポップ cx
      追加 ディー-32      一致しないので、次のエントリに進む(32バイト)
      ループ .LOOP
      jmp  FAILURE          ファイルが存在しませんでした :(
```

次のステップへ....。

ステップ3: FATの読み込み

ステップ1：スタートクラスタの取得

さて、ルートディレクトリが読み込まれ、ファイルのエントリが見つかりました。その開始クラスタを得るにはどうしたらいいのでしょうか？

- Bytes 26-27 :最初のクラス
- タ バイト 28-32：ファイルサイズ

これは見覚えがあると思います :)開始クラスタを取得するには、ファイルエントリのバイト26を参照してください。

```
mov     dx, [di + 0x001A]; diには、エントリーの開始アドレスが入ります。エントリーのバイト26(0x1A)を参照する
abl     だけです。
; Yippe--dxは開始クラスタ番号を保存するようになりました。
```

開始クラスタは、ファイルを読み込む際に重要となります。

ステップ2 : **FAT**のサイズを知る

もう一度、BIOSパラメータブロックを見てみましょう。具体的には...

bpbNumberOfFATs:	DB 2
bpbSectorsPerFAT:	DW 9

さて、ではどうやって両方のFATにあるセクタ数を調べればいいのでしょうか？FATあたりのセクタ数とセクタ数を掛け合わせるだけです :)簡単ですね。...だけど...

xor ax, ax	
movab アル, [bpbNumberOfFATs]	FATの数
le マルチ WORD [bpbSectorsPerFAT]	FATあたりのセクタ数を乗じて
; ax = FATが使用するセクタの数!	

いいえ、気にしないでください、それは簡単です^^。

ステップ3 : FATの読み込み

読むべきセクターの数がわかったのだから...読んでみてください :)

movab bx, 0x0200	ロード先のアドレス
le コール ReadSectors	FATテーブルのロード

やったーさて、FATの問題が解決したところで（完全に解決したわけではありません！）、ステージ2のロードです。

FAT - 完全な例

以下は、ブートローダーから直接取得した完全なコードです。

LOAD_FATです。	
ブートイメージの開始クラスタを保存	
movab si, msgCRLF	
le プリント	
コール	
movab dx, WORD [di + 0x001A].	
le	
movab WORD [クラスタ], dx	ファイルの最初のクラスタ
FATのサイズを計算し、"CX"に格納する。	
xor ax, ax	
movab al, BYTE [bpbNumberOfFATs].	FATの数
le	
FATの位置を計算し、"ax"に格納する。	FATが使用するセクタ
movab 55h, WORD [bpbSectorsPerFAT]	
movab 56h, WORD [bpbReservedSectors].	ブートセクターの調整
addl [bpbReservedSectors].	
FATをメモリに読み込む(7C00:0200)	
mov bx, 0x0200	ブートコードの上にFATをコピー
call ReadSectors	する。

LBAとCHS

イメージを読み込む際に必要なのは、FATを参照して各クラスタを読み込むことです。

しかし、まだ議論していない小さな問題があります。さて、FATからクラスタ番号を得ました。でも、それをどうやって使うの？

問題は、このクラスタがリニアアドレスを表しているのに対し、セクタをロードするためには、セグメント/トラック/ヘッドのアドレスが必要になることです。(インタラプト0x13)

ディスクにアクセスするには2つの方法があります。CHS (Cylinder/Head/Sector) アドレッシングとLBA (Liogical Block Addressing) のどちらかです。

LBAは、ディスク上のインデックス付きの場所を表します。最初のブロックは0で、次に1、というようになります。LBAは単にセクタがLBA0から順に番号付けされていることを表しています。これ以上の基本はありません。

LBAとCHSの変換方法を知っておく必要があります。

CHSからLBAへの変換

CHSをLBAに変換する式です。

LBA	=	(クラスタ - 2) * クラスタあたりのセクタ数
-----	---	-----------------------------

それはとても簡単なことです。)例を挙げてみましょう。

サブ	ax, 0x0002	クラスターナンバーから2を引く
xor	cx, cx	
movab	cl, BYTE [bpbSectorsPerCluster]	クラスターあたりのセクタ数
le		
マルチ	cx	マルチブライ

LBAをCHSに変換する

これは少し複雑ですが、それでも非常に簡単です。

アブソリュート・セクター	=	(LBA % セクタ/トラック) + 1
アブソリュートヘッド	=	(LBA / セクタ/トラック) % ヘッドの数
アブソリュート・トラック	=	LBA / (トラックあたりのセクタ数 * ヘッドの数)

その一例をご紹介します。

LBACHS		
です。	<pre> xor dx, dx div WORD [bpbSectorsPerTrack] inc dl mov BYTE [absoluteSector], dl </pre>	<p>dx:axの動作準備 トラックごとのセクタ数で割る 1 (オブソキュート・セクター・フォーミュラ)を追加</p>
これらのフォーラムはとても似ていますね。		
	<pre> xor dx, dx div WORD [bpbHeadsPerCylinder] mov BYTE [absoluteHead], dl mov BYTE [absoluteTrack], al ret rea t </pre>	<p>dx:axの動作準備 ヘッドの数で計算 (Absolute head formula それ以外のことは、最初の数式からすでに行われていました。 他にすることはありません。)</p>

難しいことではないと思います。)

クラスターの読み込み

さて、ステージ2の読み込みでは、まずFATからクラスタを再構築する必要があります。とても簡単です。そして、クラスタ番号をLBAに変換して、読み込めるようにします。

mov	ax,	読みたいクラスタ
pop	[cluster] bx	読み込み先のバッファ
call	ClusterLBA	クラスタをLBAに変換
xor	cx, cx	
mov	cl, [bpbSectorsPerCluster]	読めるセクター
call	ReadSectors	クラスタで読む
push	bx	

次のクラスターを取得する

これは厄介だ。

さて、**FAT**エントリの各クラスタ番号は**12ビット**であることを覚えておいてください。これが問題なのだ。**1バイト**読み込めば、クラスタ番号の一部しかコピーしていないことになります

そのため、WORD（2バイト）の値を読み取る必要があります。

しかし、ここでまた問題が発生します。12ビットの値から2バイトをコピーすることは、**次のクラスタエントリの一部**をコピーすることになります。例えば、これがあなたのFATだとします。

注：バイト単位で区切られた2進数。各12ビットのFATク
ラスタ・エントリが表示されます。

01011101 0111010 0111010 | 00111101 00111101 0111010 0011110 0011110

|
| | | | | | |
| 第1 クラスタ クラスタ 第3ク |
| ラスタ- 第2クラスタ クラスタ ----|| 2nd |
| クラスタ クラスタ 14th クラスタ |
.....

偶数クラスタは第1バイトのすべてを、第2バイトの一部を占めていることに注目してください。また、すべての奇数クラスタは、第1バイトの一部をオコピしますが、第2バイトのすべてをオコピすることに気がつきます。

さて、ここでやるべきことは、FATから2バイト(ワード)の値を読み込むことです(これが我々のクラスタ

です)。クラスタが偶数の場合は、**上位4ビット**をマスクアウトして、次のクラスタに属します。

奇数の場合は、**4ビット下**にシフトします（最初のクラスタで使用したビットを破棄します）。

次のクラスタを計算する

```
mov     ax, WORD [cluster] ; FATから現在のクラスタを特定する。
abl
```

クラスタは奇数か偶数か？2で割ってみてください。

```
movab   cx, ax              ; 現在のクラスタをコピー
le
movab   dx, ax              ; 現在のクラスタをコピー
le
```

shr	dx, 0x0001	2で割る
追加	cx, dx	の和 (3/2)ズ
movab le	bx, 0x0200	メモリ内のFATの位置
追加	bx, cx	FATへのインデックス
movab le	dx, WORD [bx].	FATから2バイトを読み込む

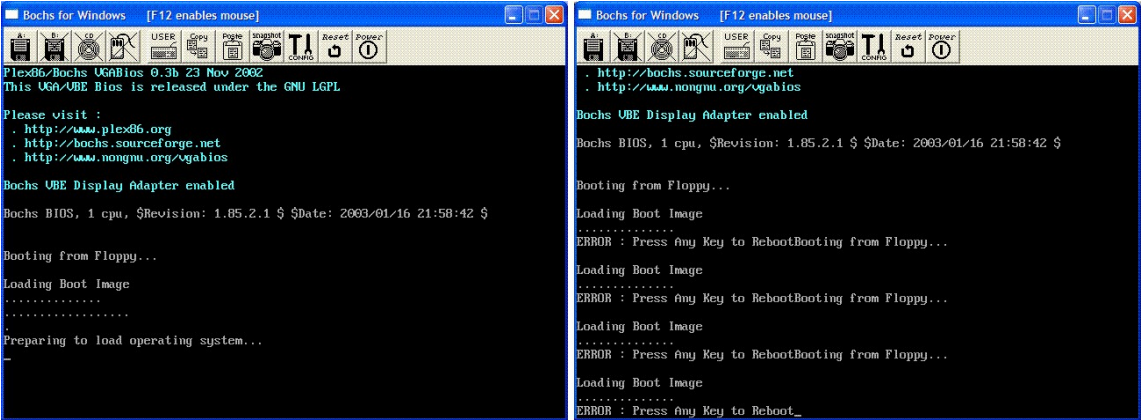

```
testax, 0x0001
jnz.ODD_CLUSTER
```

FATの各エントリは12の値であることを覚えておいてください。を表す場合は
クラスター（0x002~0xFEf）の場合、その12ビットを取得します。
次のクラスターを代表するもの

```
.EVEN_CLUSTER:
    と      dx, 0000111111111111b      下位12ビットを取る
    jmp     .DONE

.ODD_CLUSTER:
    shr     dx, 1                      上位12ビットを取る
    mov     WORD [cluster],
    cmp     dx dx, 0x0FF0             ファイルの終わりを示すテスト
    jnb     LOAD_IMAGE               まだ終わっていない、次のクラスターへ
    .DONE
```

デモ



最初のショットでは、ブートローダが**Stage 2**を正常にロードしています。ステージ2は、ロード中のオペレーティングシステムのメッセージを表示します。2つ目のショットでは、ファイルが見つからない場合のエラーメッセージが表示されます（ルートディレクトリ内）。

このデモには、このレッスンのコードのほとんど、2つのソースファイル、2つのディレクトリ、2つのバッチプログラムが含まれています。最初のディレクトリにはステージ1のプログラム（ブートローダ）があり、2番目のディレクトリにはステージ2のプログラム（STAGE2.SYS）があります。

デモのダウンロードはこちら

結論

このチュートリアルを書くのは大変でした。というのも、このような複雑なテーマを細部まで説明するのは大変ですが、その一方で、非常にわかりやすい内容にしなければなりません。うまく書けたと思います。）

もし、このチュートリアルをより良くするための提案があれば、ぜひ教えてください:)さて

...これで終わりですね。ブートローダよ、さようなら。

次のチュートリアルでは、ステージ2の構築を開始します。A20について、そしてプロテクトモードについて、さらに詳しくご紹介します。お楽しみに

次の機会まで。

マイク
BrokenThorn Entertainment社。現在、DoEとNeptuneOperating Systemを開発中です。質問

やコメントはありますか?お気軽にお問い合わせください。

あなたも記事の改善に貢献したいと思いませんか?もしそうなら、ぜひ私に教えてください。



オペレーティングシステム開発シリーズ

オペレーティングシステム開発 - システムアーキテクチャ by Mike, 2008

このシリーズは、オペレーティングシステムの開発を一から実演し、教えることを目的としています。

はじめに

歓迎します。:) 前回のチュートリアルでは、ようやくブートローダを完成させることができました。よかったです

ね。とりあえず、今のところは :) **FAT12** ファイルシステムの詳細を説明し、ステージ2の読み込み、解析、実行を見ました。

このチュートリアルは、前作の続きです。まず、**x86** アーキテクチャの詳細を見ていきます。これは、特にプロテクトモードや、プロテクトモードがどのように機能するかを理解する上で重要になります。

コンピュータがどのように動作しているのか、ビットレベルに至るまで、あらゆることを網羅していきます。起動時の **BIOS** との関係を理解するには、他のプロセッサを「起動」させることができることを覚えておく必要があります。**BIOS** はメインプロセッサでこれを行います。同じことをしてマルチプロセッサシステムをサポートすることができます。

カバーしていきます。

- **80x86** レジスター シ
- システムの構成 システム
- バス
- **Real Mode Memory Map** 命令
- が実行される仕組み **Software**
- **Ports**

ある意味では、システムアーキテクチャのチュートリアルのようなものです。しかし、ここでは **OS** 開発の観点からアーキテクチャを見ていきます。また、**アーキテクチャの中**のあらゆることを網羅します。

基本的な概念を理解することで、**プロテクトモード** をより詳細に理解することができます。次のチュートリアルでは、ここで学んだことをすべて使って、プロテクトモードに切り替えてみましょう。

楽しもうじゃないか...

プロテクトモードの世界

この言葉は誰もが聞いたことがあるのではないのでしょうか？ **Protected Mode (PMode)** は、**80286** 以降のプロセッサで利用可能な動作モードです。**PMode** は主にシステムの安定性を高めるために設計されました。

これまでのチュートリアルでご存じのように、**Real Mode** にはいくつかの大きな問題があります。ひとつは、好きな場所にバイトを書き込めることです。そのため、ソフトウェアポートやプロセッサ、あるいは自分自身が使用しているコードやデータを上書きすることができます。しかも、直接的にも間接的にも、**4,000** 通り以上の方法でこれを行うことができます。

リアルモードには**メモリ保護**がありません。すべてのデータとコードは、単一の汎用メモリブロックにダンプされます。リアルモードでは、**16** ビットのレジスターに限定されます。このため、メモリは**1MB**に制限されます。

ハードウェアレベルの**メモリ保護**や**マルチタスク**には対応していません。

おそらく最大の問題は、「リング」というものが存在しないことでした。すべてのプログラムはリング**0** レベルで実行され、すべてのプログラムがシステムを完全に制御できるようになっています。つまり、シングルタスク環境では、(**cli/hlt** のような) **1** つの命令が、注意していないと **OS** 全体をクラッシュさせてしまうのです。

これらの問題の多くは、リアルモードを詳しく説明したときに見覚えがあるはずですが。プロテクトモードは、これらの問題をすべて解決します。プロテクトモード。

- **メモリ保護機能**
- **仮想メモリ**と**TSS (Task State Switching)** をハードウェアでサポートしていること
- プログラムをインタールーティングして別のプログラムを実行するための
- ハードウェアサポート **4** つのオペレーティング・モード **リング0**、**リング1**、**リング2**、**リング3**
- **32** ビットレジスタへのアクセス 最大
- **4GB** のメモリへのアクセス

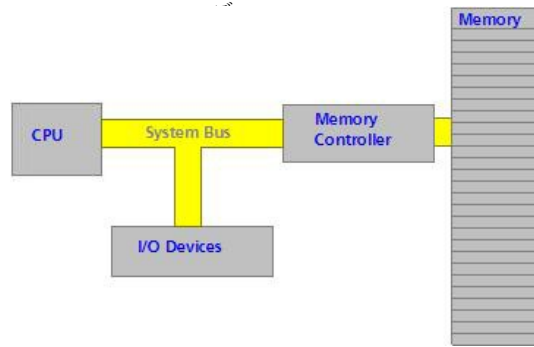
前回のチュートリアルで、アセンブリ言語の Ring について説明しました。**通常のアプリケーションが Ring 3 (Usually)** であるのに対し、我々は **Ring 0** であることを覚えておってください。通常のアプリケーションにはない特別な命令やレジスタにアクセスすることができます。このチュートリアルでは、**LGDT** 命令、独自に定義したセグメントを使用した**遠距離ジャンプ**、**プロセッサ制御レジスタ**の使用を行います。これらは、通常のプログラムでは利用できません。

システムアーキテクチャやプロセッサの動作を理解することで、この問題をより深く理解することができます。

システムアーキテクチャ

x86 ファミリーのコンピュータは、**Van Neumann Architecture** を採用しています。ヴァンヌーマン・アーキテクチャーとは、典型的なコンピュータシステムが**3** つの主要なコンポーネントを持つとする設計仕様である。

- セントラル・プロセッシング・ユニ
- ット (CPU) メモリ
- Input/Output (IO)



いくつかの重要なポイントがあります。ご存知のように、CPUはメモリからデータや命令をフェッチします。メモリコントローラは、それが存在するRAMチップとメモリセルを正確に計算する役割を担っています。このため、**The CPUはMemory Controllerと通信します。**

また、「I/Oデバイス」にも注目してください。これらはシステムバスに接続されています。**すべてのI/Oポートは、所定のメモリ位置にマッピングされています。これにより、IN命令とOUT命令を使うことができます。**

ハードウェアデバイスは、システムバスを通じてメモリにアクセスすることができます。また、何かが起こっているときに、デバイスに通知することもできます。例えば、ハードウェアデバイスのコントローラが読み取るために、メモリロケーションにバイトを書き込むと、**プロセッサはデバイスにそのアドレスにデータがあることを知らせることができます。**これは、システムバス全体のうち、**コントロールバス**という部分で行われます。これがソフトウェアとハードウェアデバイスとの基本的なやりとりです。プロテクトモードのデバイスと通信するための**唯一**の方法なので、この点は重要です。

まず、すべてを詳しく説明します。次に、それらを組み合わせて、ハードウェアレベルで命令が実行されるのを見ながら、それぞれがどのように機能するのかを学びます。ここからは、I/Oポートや、ソフトウェアとハードウェアの相互作用について説明していきます。

x86のアセンブリを経験したことのある方なら、一部、あるいはほとんどの部分は知っているはずですが、ここではほとんどのアセンブリの本では詳しく説明されていないことをたくさん取り上げます。具体的には、Ring 0プログラムに特有のことです。

システムバス

システムバスは、フロントサイドバスとも呼ばれ、CPUとマザーボード上のノースブリッジを接続します。

システムバスは、**データバス、アドレスバス、コントロールバスを組み合わせたものです。**このバスの各電子ラインは**1ビット**を表します。0 "と" 1 "を表す電圧レベルは、**TTL (Standard Transistor-Transistor Logic)** レベルに基づいています。このことを知る必要はありません。TTLは、コンピュータを構成する「**デジタル・ロジック・エレクトロニクス**」の一部である。

ご存知の通り、システムバスは3つのバスで構成されています。では、その詳細を見ていきましょう。

データバス

データバスとは、データを伝送するための一連の電子ラインのことです。データバスのサイズは、**16ライン/ビット、32ライン/ビット、64ライン/ビット**のいずれかです。電子線と1ビットの直接的な関係に注目。

つまり、**32ビットプロセッサは32ビットのデータバスを持ち、使用します。**つまり、4バイトのデータを同じように扱えるということです。このことを知っていれば、プログラムのデータサイズを気にすることができ、スピードアップにつながります。

どのように？プロセッサは、**1,2,4,8,16ビット**のデータをデータバスのサイズに合わせて0で埋める必要があります。大きなデータは、プロセッサがデータバス上で正しくバイトを送信できるように、分割（およびパディング）する必要があります。**データバスのサイズに合わせてデータを送信すると、余分な処理が行われないため高速になります。**

例えば、データタイプが**64ビット**で、データバスが**32ビット**の場合を考えてみましょう。最初の**クロックサイクル**では、最初の**32ビット**だけがデータバスを通してメモリコントローラに送られます。2回目の**クロックサイクル**では、プロセッサは最後の**32ビット**を参照します。**注：データタイプが大きくなるほど、より多くのクロックサイクルが必要になることに注意してください。**

一般的に、「**32ビットプロセッサ**」「**16ビットプロセッサ**」などの用語は、データバスのサイズを指します。つまり、「**32ビットプロセッサ**」は、**32ビット**のデータバスを使用しています。

アドレスバス

プロセッサやI/Oデバイスがメモリを参照する必要があるときは、アドレスバスにアドレスを配置します。さて、**メモリアドレス**がメモリ上の位置を表すことは周知の通りです。しかし、これは抽象的なものです。

「メモリアドレス」とは、**メモリーコントローラが使用する番号のこと**です。そうなんです。メモリコントローラは、このバスから数字を受け取り、それをメモリの位置として解釈します。メモリコントローラは、**各RAMチップのサイズを知っている**ので、**正確なRAMチップとその中のバイトオフセットを簡単に参照することができます。**メモリセル**0**から始まって、メモリコントローラはこのオフセットを**必要なアドレス**としてインタープリートします。

アドレスバスは、**コントロールユニット (CU)**、**I/Oコントローラ**を介してプロセッサに接続されています。**コントロールユニット**はプロセッサの内部にあるので、後ほどご紹介します。**I/O Controller**は、ハードウェアデバイスとのインターフェースを制御します。これについては後述します。

データバスと同様に、各電子ラインは**1つのビット**を表します。1ビットには2つの値しかないので、**CPUがアクセスできるアドレスは正確に2^n個あります。**したがって、アドレスバスのビット数やライン数は、**CPUがアクセスできる最大のメモリを表しています。**

8080から80186までのプロセッサは、それぞれ**20ライン/ビット**のアドレスバスを持っていました。80286と80386では**24本/ビット**、80386+では**32本/ビット**となっています。

x86ファミリー全体が、すべての古いプロセッサと互換性があるように設計されていることを覚えておってください。これがリアルモードで起動する理由です。これまでのプロセッサアーキテクチャでは、**0行目から19行目までの20本のアドレスライン**にしかアクセスできなかったため、**1MBに制限されていた**。

これは私たちにとって重要なことで、この制限はまだ私たちに適用されます。必要なのは、**20番目のアドレスラインからのアクセスを可能にすること**です。これにより、**OSは4GB以上のメモリにアクセスできるようになります。**詳しくは後述します。

コントロールバス

さて、データバスにデータを置き、アドレスバスでメモリのアドレスを参照することはできました。しかし、このデータをどうすればいいのでしょうか？メモリから読み出しているのか？それとも、データを書き込むのか？

コントロールバスは、デバイスが何をしようとしているかを表す一連のライン/ビットです。例えば、プロセッサはREADビットまたはWRITEビットを設定して、アドレスバスに格納されているメモリロケーションからデータバスのデータを読み書きしたいことをメモリコントローラに知らせます。

コントロールバスは、プロセッサがデバイスに信号を送ることも可能にします。これは、デバイスに注意が必要であることを知らせるものです。例えば、アドレスバスからメモリの場所を読み出す必要があるとします。これにより、デバイスに必要なことを知らせることができます。これは、**I/Oソフトウェアのポートで重要です**。

もちろん、システムバスはハードウェアデバイスに直接接続されているわけではありません。**I/Oコントローラ**に接続されており、I/Oコントローラがデバイスに信号を送ります。

以上です。

これがシステムバスのすべてである。システムバスは、プロセッサ（**コントロールユニット（CU）** 経由）やI/Oデバイス（**I/Oコントローラ**経由）から**メモリコントローラ**に至るまで、メモリへのアクセスや読み出しを行うための経路であり、メモリコントローラは正確なRAMチップを計算し、アクセスしたいメモリセルを見つけ出す役割を担っています。

"コントローラ"・・・この言葉はよく耳にしますね。理由は後述します。

メモリコントローラ

メモリコントローラは、マザーボード上のシステムバス（FSB）と物理的なRAMチップとの間の主要なインターフェースです。**コントローラ**という言葉を見たことがあると思います。そもそもコントローラとは何でしょうか？

コントローラ

コントローラは、ハードウェアの基本的な制御機能を提供します。また、ハードウェアとソフトウェアの間の**基本的なインターフェースを提供します**。これは私たちにとって重要なことです。プロテクトモードでは、**利用できる割り込みがない**ことを覚えておいてください。ブートローダでは、ハードウェアとの通信にいくつかの割り込みを使用しました。**プロテクトモードでこれらの割り込みを使用すると、トリプルフォールトが発生します**。どうしたらいいのでしょうか？

そのためには、ハードウェアと直接通信する必要があります。そのためには、コントローラを使用します。（コントローラの仕組みについては、後ほどI/Oサブシステムを取り上げる際に詳しく説明します。）

メモリコントローラ

メモリコントローラは、ソフトウェアを使ってメモリの読み出しと書き込みを行う手段を提供します。また、メモリーコントローラは、RAMチップが情報を保持できるように、常にリフレッシュする役割も担っています。

メモリコントローラは、**マルチプレクサ回路とデマルチプレクサ回路**により、アドレスバス内のアドレスに対応するRAMチップとその位置を正確に選択します。

ダブルデータレート（DDR）コントローラ

DDRコントローラは、**システムクロック**のパルスを利用してメモリの読み書きを可能にするDDR SDRAMのリフレッシュに使用されます。

デュアルチャネルコントローラ

デュアルチャネルコントローラは、DRAMデバイスを2つの小さなバスに分離し、2つのメモリロケーションを同時に読み書きできるようにしたものです。これにより、RAMへのアクセス速度が向上します。

メモリコントローラの結論

メモリコントローラは、アドレスバスに入力されたアドレスを受け取ります。しかし、メモリコントローラにメモリの読み書きを指示するにはどうすればいいのでしょうか。また、どこからデータを取得するのでしょうか？メモリを読み出す場合、**プロセッサはControl BusにReadビットをセットします**。同様に、メモリを書き込むときには、**プロセッサはControl BusにWriteビットをセットします**。

コントロールバスは、他のデバイスがバスをどのように使用するかをプロセッサが制御できることを覚えておいてください。

メモリコントローラが使用するデータは**Data Bus**の中にあります。使用するアドレスは、アドレスバスの中にあります。

リーディングメモリ

メモリを読み出す場合、プロセッサは読み出し先の絶対アドレスをアドレスバスに配置します。その後、プロセッサは**READ**コントロールラインを設定します。

これでメモリコントローラが制御できるようになりました。コントローラは、**マルチプレクサ回路**を使って絶対アドレスを物理的なRAMの位置に変換し、データをデータバスに配置します。そして、**READ**ビットを0にリセットし、**READY**ビットをセットします。

プロセッサは、データがデータバスに入ったことを知ります。このデータをコピーして、残りの命令を実行する.....おそらく**BX**に格納するのではないだろうか？

ライティングメモリ

メモリの書き方も似ています。

まず、プロセッサは、メモリアドレスをアドレスバスに配置します。次に、書き込むデータをデータバスに配置します。そして、**Control Bus**に**WRITE**ビットをセットします。

これにより、メモリコントローラは、データバスのデータをアドレスバスの絶対アドレスに書き込むことができます。書き込みが完了すると、メモリコントローラは**WRITE**ビットをリセットし、**Control Bus**の**READY**ビットをセットします。

結論

私たちは、ソフトウェアを使って直接メモリーコントローラと通信するのではなく、間接的に通信しているのです。**メモリを読み書きするときには、必ず「メモリコントローラ」を使います**。これが、我々のソフトウェアとメモリーコントローラ／RAMチップのハードウェアとの間のインターフェースである。

さて、次はI/Oサブシステムを見てみましょうか。そうだ。あの**1337マルチブレイクサ**回路は？あれはメモリーコントローラーの中にある物理的な電子回路です。その仕組みを理解するには、**デジタル・ロジック・エレクトロニクス**を知らなければならない。これは私たちには理解できないことなので、ここでは説明しません。もっと詳しく知りたい方は、**Google!**

I/Oサブシステム

I/O SubSystemは、単に**Port I/O**を表す。これは、ソフトウェアとハードウェアコントローラーの間のインターフェースを提供する基本システムである。

もっとよく見てみると...

ポート

ポートは、単純に2つの機器間のインターフェースを提供するものです。ポートには2種類あります。**ハードウェアポート**と**ソフトウェアポート**です。

ハードウェアポート

ハードウェアポートは、2つの物理デバイス間のインターフェースを提供します。このポートは通常、ある種の接続デバイスです。これには、以下のものが含まれますが、これらに限定されません。**シリアルポート、パラレルポート、PS/2ポート、1394、FireWire、USBポート**など。

これらのポートは通常、一般的なコンピュータシステムの側面/背面/前面にあります。

さて...ポートを見たい場合は、コンピュータに接続されているすべての線をたどってください。お願いだから、ジークスのためにも、これが何をするものなのか聞かないでください。マジで！？

一般的な電子機器では、これらのポートのピンは、ハードウェアデバイスに応じて異なることを示す信号を運びます。これらのピンは、システムバスと同じように、待つてください.....。ビットです。各ピンは1つのビットを表します。そう、それです。

ハードウェアのポートには、一般的に「オス」と「メス」の2つの分類があります。オス型ポートは、コネクタからピンが出ている接続です。メスポートはその逆である。ハードウェアポートは、コントローラを介してアクセスします。詳しくは後述しますが...

ソフトウェアポート

これは私たちにとって非常に重要なことです。これはハードウェアへのインターフェースです。**ソフトウェアポート**は数字です。それだけである。この番号は、ハードウェアのコントローラを表しています...。そんな感じです。

複数のポート番号が同じコントローラを表すことがあることをご存知でしょうか。その理由は？**メモリーマップドI/O**です。基本的な考え方は、あるメモリアドレスを指定してハードウェアに通信するというものです。**ポート番号はこのアドレスを表します。**.....もう一回、ちょっとだけ。のようなものです。アドレスの意味は、デバイスの特定のレジスタ、または制御レジスタを表すことができます。詳しくは後

で見ましょう。

メモリーマッピング

x86アーキテクチャでは、プロセッサは特定のメモリーロケーションを使って特定のものを表現します。

例えば、**0xA000:0**というアドレスは、**ビデオカードのVRAMの開始点を表しています**。この場所にバイトを書き込むことで、ビデオメモリ内の内容を変更し、画面に表示される内容を変更することができます。

例えば、FDC (Floppy Drive Controller) のレジスタなどです。どのアドレスが何を表しているのかを理解することは、私たちにとって非常に重要なこと

となのです。

x86リアルモードメモリマップ

一般的なx86リアルモードのメモリマップです。

- **0x00000000 - 0x000003FF** - リアルモードインタラプトベクターテーブル
- **0x00000400 - 0x000004FF** - BIOSデータエリア
- **0x00000500 - 0x00007BFF** - 未使用
- **0x00007C00 - 0x00007DFF** - 当社ブートローダ
- **0x00007E00 - 0x0009FFFF** - 未使用
- **0x000A0000 - 0x000BFFFF** - ビデオRAM (VRAM) メモリ
- **0x000B0000 - 0x000B7777** - モノクロ・ビデオメモリ
- **0x000B8000 - 0x000BFFFF** - カラー・ビデオメモリ
- **0x000C0000 - 0x000C7FFF** - ビデオROM BIOS
- **0x000C8000 - 0x000EFFFF** - BIOSシャドウ領域
- **0x000F0000 - 0x000FFFFF** - システムBIOS

注：上記のデバイスをすべてリマップして、メモリの異なる領域を使用することが可能です。これは、BIOSのPOSTがデバイスを上の表にマッピングするために行うものです。

なるほど、これはかっこいいですね。これらのアドレスはそれぞれ異なるものを表しているの、特定のアドレスに読み書きすることで、コンピュータのさまざまな部分から簡単に情報を得る（変更する）ことができます。

例えば、**INT 0x19**について話したことを覚えていますか？0x0040:0x0072に0x1234という値を書き込み、0xFFFF:0にジャンプすると、実質的にコンピュータをウォームリブートすることができると説明しました。(Windowsのctrl+alt+delに似ています。) seg:offsetアドレスモードと絶対アドレスの変換を思い出して、**0x0040:0x0072**を絶対アドレス**0x000000472**(BIOSデータエリア内の1バイト)に変換します。

他の例としては、テキスト出力があります。しかし、**0x000B8000**に2バイト書き込むことで、テキストモードのメモリにあるものを効果的に変更することができます。これは表示時に常に更新されるので、実質的に画面に文字が表示されることになります。かっこいいでしょ？

話をポートマッピングに戻しましょう。このテーブルについては、後ほど詳しく説明します。

ポートマッピング - メモリマップドI/O

「ポートアドレス」とは、各コントローラが開き取る特別な番号のことです。起動時には、**ROM BIOS**がこれらのコントローラデバイスに異なる番号を割り当てます。

プライマリプロセッサを起動し、0xFFFF:0にBIOSプログラムをロードします(これを覚えていますか？ 前節の表と比較してみてください)。

ROM BIOSは、この番号をコントローラごとに割り当て、コントローラが自分を識別できるようにします。これにより、BIOSは割り込みベクトルテーブルを設定し、この特別な番号を使ってハードウェアと通信します。

プロセッサは、I/Oコントローラと連携する際に、同じシステムバスを使用します。**プロセッサは、あたかもメモリを読み込むかのように、特別なポート番号をアドレスバスに入れます。**また、制御バスにもREADまたはWRITEラインを設定します。これは素晴らしいことですが、問題があります。プロセッサは、メモリの書き込みとコントローラへのアクセスをどのように区別しているのでしょうか？

プロセッサは、コントロールバス上にもう一つのライン--I/O ACCESSラインを設定する。**このラインが設定されると、I/Oサブシステム内のI/Oコントローラは、アドレスバスを監視します。アドレスバスがデバイスに割り当てられている番号に反応した場合、そのデバイスはデータバスから値を取り、それを処理する。**メモリコントローラは、このラインがセットされている場合、いかなる要求も無視する。つまり、ポート番号が割り当てられていなければ、まったく何も起こりません。どのコントローラも動作せず、メモリコントローラはそれを無視します。

では、このポートアドレスを見てみましょう。これは非常に重要です。これはプロテクトモードのハードウェアと通信するための ***唯一の* 方法です！**：

警告このテーブルは大きいです。

デフォルトのx86ポートアドレスの割り当て				
アドレス 範囲	最初のQWORD	2つ目のQWORD	3つ目のQWORD	4つ目のQWORD
0x000-0x00F	DMAコントローラチャンネル0～3			
0x010-0x01F	システム 使用			
0x020-0x02F	割り込みコントローラ1	システム利用		
0x030-0x03F	システム 使用			
0x040-0x04F	システム タイマー	システム利用		
0x050-0x05F	システム 使用			
0x060-0x06F	キーボード／PS2 Moude（ポート0x60 スピーカー（0x61	キーボード／PS2マウス（0x64	システム利用	
0x070-0x07F	RTC/CMOS/NMI (0x70, 0x71)	DMAコントローラチャンネル0～3		
0x080-0x08F	DMAページレジスタ0-2 (0x81 - 0x83)	DMAページレジスタ3（0x87	DMAページレジスタ4-6 (0x89-0x8B)	DMAページレジスタ7（0x8F
0x090-0x09F	システム 使用			
0x0A0-0x0AF	割り込みコントローラ2 (0xA0-0xA1)	システム利用		
0x0B0-0x0BF	システム 使用			
0x0C0-0x0CF	DMAコントローラチャンネル4-7（0x0C0-0x0DF）、バイト1-16			
0x0D0-0x0DF	DMAコントローラチャンネル4-7（0x0C0-0x0DF）、バイト16-32			
0x0E0-0x0EF	システム 使用			
0x0F0-0x0FF	浮動小数点演算ユニット（FPU/NPU/Mah Coprocessor			
0x100-0x10F	システム 使用			
0x110-0x11F	システム 使用			
0x120-0x12F	システム 使用			
0x130-0x13F	SCSI Host Adapter (0x130-0x14F), バイト 1-16			
0x140-0x14F	SCSI Host Adapter (0x130-0x14F), バイト17-32		SCSI Host Adapter (0x140-0x15F), バイト 1-16	
0x150-0x15F	SCSI Host Adapter (0x140-0x15F), バイト17-32			
0x160-0x16F	システム利用		第4世代のIDEコントローラ、マスター・スレーブ	
0x170-0x17F	セカンダリIDEコントローラ、マスタードライブ		システム利用	
0x180-0x18F	システム 使用			
0x190-0x19F	システム 使用			
0x1A0-0x1AF	システム 使用			
0x1B0-0x1BF	システム 使用			
0x1C0-	システム 使用			

0x1CF	ズ		
0x1D0-0x1DF	システム利用		
0x1E0-0x1EF	システム利用	3rdary IDEコントローラ、マスター・スレーブ	
0x1F0-0x1FF	プライマリIDEコントローラ、マスター・スレーブ	システム利用	
0x200-0x20F	ジョイスティックポート	システム利用	
0x210-0x21F	システム利用		
0x220-0x22F	サウンドカード		
	非NE2000ネットワークカード	システム利用	
0x230-0x23F	SCSI Host Adapter (0x220-0x23F), バイト 17-32)		
0x240-0x24F	サウンドカード		
	非NE2000ネットワークカード	システム利用	
	NE2000 ネットワークカード (0x240-0x25F) 1-16 バイト		
0x250-0x25F	NE2000 ネットワークカード (0x240-0x25F) 17-32 バイト		
0x260-0x26F	サウンドカード		
	非NE2000ネットワークカード	システム利用	
	NE2000 ネットワークカード (0x240-0x27F) バイト 1-16		
0x270-0x27F	システム利用	プラグアンドプレイシステム機器	LPT2 - 2番目のパラレルポート
	システム利用		LPT3 - 第3パラレルポート (モノクロシステム
	NE2000ネットワークカード(0x260-0x27F) 17-32バイト		
0x280-0x28F	サウンドカード		
	NE2000以外のネットワークカード	システム利用	
	NE2000 ネットワークカード (0x280-0x29F) バイト 1-16		
0x290-0x29F	NE2000ネットワークカード(0x280-0x29F) 17～32バイト目		
0x2A0-0x2AF	NE2000以外のネットワークカード	システム利用	
	NE2000 ネットワークカード (0x280-0x29F) バイト 1-16		
0x2B0-0x2BF	NE2000ネットワークカード(0x280-0x29F) 17～32バイト目		
0x2C0-0x2CF	システム利用		
0x2D0-0x2DF	システム利用		
0x2E0-0x2EF	システム利用		COM4 - 4番目のシリアルポート
0x2F0-0x2FF	システム利用		COM2 - 2番目のシリアルポート
0x300-0x30F	サウンドカード/MIDIポート	システム利用	
	NE2000以外のネットワークカード	システム利用	
	NE2000 ネットワークカード (0x300-0x31F) 1-16 バイト		
0x310-0x31F	NE2000ネットワークカード(0x300-0x32F) 17-32バイト		
0x320-0x32F	サウンドカード/MIDIポート (0x330, 0x331	システム利用	
	NE2000 ネットワークカード (0x300-0x31F) 17-32 バイト		
	SCSI ホストアダプタ (0x330-0x34F) バイト 1-16		
0x330-0x33F	サウンドカード/MIDIポート	システム利用	
	NE2000以外のネットワークカード	システム利用	
	NE2000 ネットワークカード (0x300-0x31F) 1-16 バイト		
0x340-0x34F	SCSI ホストアダプタ (0x330-0x34F) バイト 17-32		
	SCSI Host Adapter (0x340-0x35F) バイト 1-16		
	NE2000以外のネットワークカード	システム利用	
	NE2000 ネットワークカード (0x340-0x35F) 1-16 バイト		
0x350-0x35F	SCSI ホストアダプタ (0x340-0x35F) バイト 17-32		
	NE2000 ネットワークカード (0x300-0x31F) 1-16 バイト		
	テープアクセラレータカード (0x360	システム利用	クォータナリーIDEコントローラ (スレーブドライブ) (0x36E-0x36F

0x360-0x36F	NE2000以外のネットワークカード	システム利用	ズ
	NE2000 ネットワークカード (0x300-0x31F) 1-16 バイト		
0x370-	テープアクセラレータカード	セカンダリIDEコントローラ (スレーブ	LPT1 - 最初のパラレルポート (カラーシステム

0x37F	(0x370)		ドライ	ズ	
	システム利用		フ)		LPT2 - 2番目のパラレルポート (モノクロームシステム)
	NE2000 ネットワークカード (0x360-0x37F) バイト				
0x380-0x38F	1-16 システム利用		サウンドカード (FMシンセサイザ		システム利用
0x390-0x39F	システム利用				
0x3A0-0x3AF	システム利用				
0x3B0-0x3BF	VGA/モノクロビデオ				LPT1 - 最初のパラレルポート (モノクロシステム
0x3C0-0x3CF	VGA/CGAビデオ				
0x3D0-0x3DF	VGA/CGAビデオ				
0x3E0-0x3EF	テープアクセラレータカード (0x370	システム利用	COM3 - 3番目のシリアルポート		
	システム利用			3rdary IDEコントローラ (スレーブドライブ) (0x3EE-0x3EF)	
0x3F0-0x3FF	フロッピーディスクコントローラ		COM1 - 最初のシリアルポート		
	テープアクセラレータカード(0x3F0)	プライマリIDEコントローラ (スレーブドライブ) (0x3F6-0x3F7)	COM2 - 2番目のシリアルポート		

この表は完全なものではなく、間違いがないことを願っています。時間が経ち、より多くのデバイスが開発されるにつれ、この表に追加していきます。

上の表にあるように、これらのメモリ範囲はすべて特定のコントローラで使用されます。ポートアドレスの正確な意味は、コントローラによって異なります。制御レジスタやステータスレジスタなど、あらゆるものが対象となります。これは、とても残念なことです。

上記の表をプリントアウトすることを強くお勧めします。ハードウェアとの通信のたびに参照する必要があります。

私が表を更新した場合は、(チュートリアル最初)に更新します。そうすれば、表をもう一度印刷して、誰もが最新のコピーを手にすることができます。

これらを踏まえて、まとめてみると...

INとOUTの指示

x86プロセッサには、ポートI/Oに使用する2つの命令があります。それがINとOUTです。

これらの命令は、デバイスと通信したいことをプロセッサに伝えます。これにより、プロセッサはコントロールバスのI/O DEVICEラインを確実に設定します。

それでは、キーボードコントローラの入力バッファから読み込めるかどうか、実際に試してみましょう。

上記のポート表を見ると、キーボードコントローラはポートアドレス0x60から0x6Fにあることがわかります。表を見ると、ポートアドレス0x60から始まる1つ目のQWORDと2つ目のQWORDがキーボードとPS/2マウス用であることがわかります。最後の2つのQWORDはシステム用なので、無視します。

さて、キーボードコントローラはポート0x60から厳密にはポート0x68にマッピングされています。これは素晴らしいことですが、私たちにとってどんな意味があるのでしょうか？これはデバイス固有のものでしょね？

今回のキーボードでは、ポート0x60がコントロールレジスタ、ポート0x64がステータスレジスタです。前にも言ったけど、これらの用語はもっと色々な文脈で出てくると思うよ。ステータスレジスタのビット1がセットされていれば、データは入力バッファの中にあります。つまり...CONTROLレジスタをREADに設定すると、入力バッファの内容をどこかにコピーすることができます...

```
oop:in      WaitL   al, 64h          ; ステータスレジスタの値の取得
           として   al, 10b         ステータス・レジスタのビット1のテスト
           jz      WaitLoop        ステータスレジスタのビットがセットされていない場合は、バッファにデータが入って
                                   いません。
           で       アル, 60H       ; Its set--バッファ (Port 0x60)からバイトを取得し、それを格納する。
```

これこそが、ハードウェア・プログラミングとデバイス・ドライバの基本です。

IN命令では、プロセッサは0x64のようなポートアドレスをアドレスバスに入れ、コントロールバスにI/O DEVICEラインを設定し、続いてREADラインを設定します。ROM BIOSで0x60に割り当てられたデバイス (この場合、キーボードコントローラのステータスレジスタ) は、READラインがセットされているので、読み出し動作であることがわかります。そこで、キーボードレジスタ内のある場所からデータバスにデータをコピーし、コントロールバスのREADラインとI/O DEVICEラインをリセットして、READYラインをセットします。これで、プロセッサはデータバスから読み込まれたデータを手に入れたことになります。

OUT命令も同様です。プロセッサは書き込まれるバイトをデータバスにコピーします (データバス幅に合わせてゼロ拡張)。そして、コントロールバスのWRITEラインとI/O DEVICEラインを設定します。そして、ポートアドレス (例えば0x60) をアドレスバスにコピーします。I/O DEVICEラインがセットされているということは、すべてのコントローラにアドレスバスを監視するように指示する信号である。アドレスバス上の数字が、割り当てられた数字と一致すれば、そのデバイスはそのデータに基づいて動作します。この例では、キーボードコントローラです。キーボードコントローラは、コントロールバスにWRITEラインが設定されているので、その操作がWRITEであることを知っています。そこで、データバス上の値を、ポートアドレス0x60が割り当てられている制御レジスタにコピーします。キーボード・コントローラは、WRITEラインとI/O DEVICEラインをリセットし、コントロール・バスにREADYラインを設定して、プロセッサは再び制御を開始します。

ポートマッピングとポートI/Oは非常に重要です。プロテクトモードのハードウェアと通信するための唯一の方法です。覚えておいてほしいのは、インタラプトは書くまで利用できないということです。割り込みを書くには、入力や出力などのハードウェアのルーチンと一緒に、ドライバを書く必要があります。そのためには、ハードウェアに直接アクセスする必要があります。慣れないうちは、少し練習してから、この章を読み直してください。何か質問があれば、私に教えてください。

ザ・プロセッサ

80x86のほとんどの命令は、どのプログラムでも実行できます。しかし、中にはカーネルレベルのソフトウェアだけがアクセスできる命令もあります。そのため、これらの命令の中には、読者にとってなじみのないものもあるかもしれません。私たちはこれらの命令のほとんどを使用する必要がありますので、これらを理解することは重要です。

Privileged Level (Ring 0) 命令	
インストラクション	説明
LGDT	GDTRにGDTのアドレスをロードする
LLDT	LDTのアドレスをLDTRにロードする
LTR	タスクレジスタをTRにロードする
MOV コントロールレジスタ	データをコピーしてコントロールレジスタに格納
LMSW	新しいマシンのロード ステータス WORD
CLTS	コントロール・レジスタのタスク・スイッチ・フラグをクリア CR0
MOV デバッグレジスタ	データをコピーしてデバッグ用レジスタに格納
INVD	ライトバックしないでキャッシュを無効にする
INVLPG	TLBエントリの無効化
WBINVD	ライトバックでキャッシュを無効にする
HLT	ハルトプロセッサ
RDMSR	モデル・スペシフィック・レジスター (MSR) の読み出し
WRMSR	MSR (Model Specific Registers) の書き込み
RDPMC	パフォーマンスモニタリングカウンターの読み込み
RDTSC	読み取り時間 スタンプカウンタ

これらの説明を理解できなくても心配しないでください。このシリーズの中で、必要に応じてそれぞれの説明をしていきます。

x86プロセッサには、現在の状態を保存するためのさまざまな**レジスタ**があります。ほとんどのアプリケーションがアクセスできるのは、**一般フラグ**、**セグメントフラグ**、および**イーフラグ**のみです。その他のレジスタは、カーネルのようなRing 0プログラムに固有のもので、

これらのレジスタの多くは、リアルモードのリング0プログラムでしか利用できません。これには非常に大きな理由があります。これらのレジスタの多くは、プロセッサ内の多くの状態に影響を与えます。これらのレジスタを誤って設定すると、簡単にCPUがトリプルフォールトになります。それ以外にも、CPUが誤動作する可能性もあります。(特に、TR4,TR5,TR6,TR7の使用について)

これらの特殊なレジスタを知る必要があるので、詳しく見ていきましょう。

また、いくつかのレジスタは文書化されていないことに注意してください。このため、リストに掲載されていないレジスタもあるかもしれません。もしご存知でしたら、追加できるようお知らせください。）

これらは32ビットのレジスタで、ほとんどすべての目的に使用することができます。しかし、それぞれのレジスタには特別な目的もあります。

- これらの32ビットのレジスタは、それぞれ2つの部分を持っています。高次のワードと低次のワードです。高次ワードは上位16ビット。低次ワードは下位16ビットです。

上位16ビットには、特別な名前はありません。しかし、下位16ビットには特別な名前があります。これらの名前には、上位8ビットには「H」が、下位8ビットには「L」が付けられています。

```

+--- ah ---+--- al ---+
+-----+-----+
+-----+-----+
|         |         |

```

www.brokenthorn.com/Resources/OSDev7.html -----EAX lower 32 bits----- | -- 32ビットのプロセッサでのみ利用可能です。

|
|----- RAX Complete 64 bits-----| -- 64ビットのプロセッサでのみ利用可能です。

これは何を意味するのでしょうか？**AH**と**AL**は**AX**の一部であり、その**AX**は**EAX**の一部である。したがって、これらの名前のいずれかを変更すると、同じレジスタ（**EAX**）を変更することになります。

これにより、64ビットマシンでは、RAXが変更されま
す。上記はBX、CX、DXにも当てはまります。

汎用レジスタは、リング0からリング4まで、あらゆるプログラムの中で使用することができます。基本的なアセンブリ言語なので、その仕組みはすでに知っているものとします。

セグメントレジスター

セグメントレジスターは、リアルモードで現在のセグメントアドレスを変更します。これらはすべて16ビットです。

- CS - コード・セグメントのセグメント・アド
- レス DS - データ・セグメントのセグメント
- ・アドレス ES - エクストラ・セグメントの
- セグメント・アドレス SS - スタック・セグ
- メントのセグメント・アドレス FS - ファー
- ・セグメント・アドレス
- GS - 汎用レジスタ

覚えておいてください。リアルモードでは、セグメント：オフセットのメモリアドレッシングモデルを採用しています。セグメントアドレスは、セグメントレジスタに格納されます。また、BP、SP、BXなどの別のレジスタには、オフセットアドレスが格納されます。

これは通常、次のように表現されます。DS:SI、DSはセグメントアドレス、SIはオフセットアドレスです。

セグメントレジスターは、リング0からリング4まで、あらゆるプログラムで使用することができます。基本的なアセンブリ言語なので、その仕組みはすでに知っているものとします。

インデックスレジスター

x86では、メモリにアクセスする際に役立ついくつかのレジスタを使用しています。

- SI - ソース・インデックス
- DI - デスティネーション
- インデックス BP - ベース
- ポインタ
- SP - スタックポインタ

32ビットのプロセッサでは、これらのレジスタは32ビットで、ESI、EDI、EBP、ESPと呼ばれています。

64ビットのプロセッサでは、各レジスタは64ビットの大きさと、RSI、RDI、RBP、RSPという名前がついています。

16ビットレジスターは32ビットレジスターのサブセットであり、64ビットレジスターのサブセットでもあります。

スタックポインタは、特定の命令が実行されるたびに、自動的に一定のバイト数を増減させます。このような命令には、push*、pop*命令、ret/iret、call、syscallなどがあります。

Cプログラミング言語は、実際にはほとんどの言語でスタックを使用します。Cが正しく動作するためには、スタックを適切なアドレスに設定する必要があります。また、覚えておいてください。スタックは*下向き*に成長します！

命令ポインター/プログラムカウンター

IP (Instruction Pointer) レジスタは、現在実行中の命令の現在のオフセットアドレスを格納しています。覚えておいてください。これはオフセットアドレスであり、絶対アドレスではありません。

IP (Instruction Pointer) は、PC (Program Counter) と呼ばれることもあります。32

ビットのマシンでは、IPは32ビットの大きさと、EIPという名前を使っています。

64ビットのマシンでは、IPは64ビットの大きさと、RIPという名前を使っています。

インストラクション・レジスタ

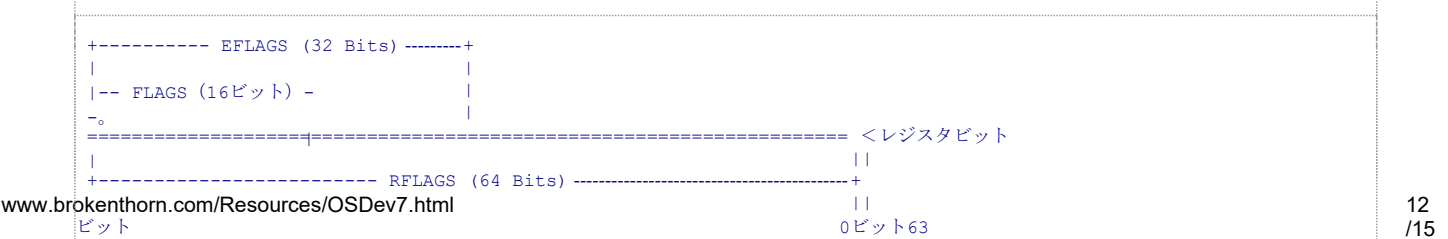
通常の方法ではアクセスできない、プロセッサ内部のレジスタです。プロセッサのコントロールユニット（CU）内のインストラクションキャッシュに格納されています。プロセッサ内部で使用するためにマイクロインストラクションに変換されている現在の命令が格納されています。詳しくは「プロセッサ・アーキテクチャ」をご覧ください。

EFLAGSレジスタ

EFLAGSレジスタは、x86プロセッサのステータスレジスタです。これは、現在のステータスを決定するために使用されます。これまでに何度も使用してきました。簡単な例：JC、JNC、JB、JNB 命令

ほとんどの命令はEFLAGSレジスタを操作して、条件をテストできるようになっています（例えば、値が他より低かったり高かったりした場合）。

EFLAGSは、FLAGSレジスタで構成されます。同様に RFLAGS は EFLAGS と FLAGS で構成されます。



FLAGSレジスタのステータスビット		
ビット数	アブリメーション	説明
0	CF	キャリーフラグ - ステータスビット
1		予約
PF	CF	パリティフラグ
3		予約
4	AF	アジャストフラグ - ステータスビット
5		予約
6	ZF	ゼロフラグ - ステータスビット
7	SF	サインフラグ - ステータスビット
8	TF	トラップフラグ (シングルステップ) - システムフラグ
9	IF	インタラプトイネーブルフラグ - システムフラグ
10	DF	方向フラグ - コントロールフラグ
11	OF	オーバーフローフラグ - ステータスビット
12-13	IOPL	I/O Privilege Level (286+ Only) - コントロールフラグ
14	NT	ネストタスクフラグ (286+のみ) - コントロールフラグ
15		予約
EFLAGSレジスタのステータスビット		
ビット数	アブリメーション	説明
16	RF	レジャーフラグ(386+のみ) - コントロールフラグ
17	VM	v8086モードフラグ(386+のみ) - 制御フラグ
18	AC	アライメントチェック (486SX+のみ) - 制御フラグ
19	VIF	仮想割込みフラグ (Pentium+のみ) - 制御フラグ
20	VIP	仮想インタラプトペンディング (Pentium+のみ) - 制御フラグ
21	ID	識別 (Pentium+のみ) - 制御フラグ
22-31		予約
RFLAGS レジスタのステータスビット		
ビット数	アブリメーション	説明
32-63		予約

IO Privilege Level (IOPL) は、特定の命令を使用するために必要な現在のリングレベルを制御します。例えば、**CLI**、**STI**、**IN**および**OUT**命令は、現在のPrivilege LevelがIOPLと同等以上の場合にのみ実行されます。そうでない場合は、**GPF (General Protection Fault)** がプロセッサによって生成されます。

ほとんどのOSでは、IOPFを0または1に設定しています。これは、カーネルレベルのソフトウェアのみがこれらの命令を使用できることを意味します。これは非常に良いことです。結局のところ、アプリケーションがCLIを発行すれば、事実上、カーネルの実行を停止させることができます。

ほとんどの操作では、**FLAGS**レジスタを使用するだけで十分です。**RFLAGS**レジスタの最後の32ビットはnull、null、non-existenceであり、見る楽しみのために存在していることに注意してください。つまり.....そうです。もちろん、高速化のためですが、多くのバイトが無駄になります。そうですね。

この表はサイズが大きいため、プリントアウトして後で参照することをお勧めします。

テストレジスター

x86ファミリーでは、テスト用にいくつかのレジスタを使用しています。これらのレジスターの多くは文書化されていません。x86シリーズでは、これらのレジスターは**tr4, tr5, tr6, tr7**です。

TR6はコマンドテストに、**TR7**はテストデータレジスタに最もよく使われます。これらにアクセスするには、**MOV**命令を使用します。これらは**pmode**と**real mode**の両方のリング**0**でのみ利用可能です。それ以外の方法でアクセスすると、**GPF (General Protection Fault)** が発生し、トリプルフォールトになります。

デバッグレジスター

これらのレジスタは、プログラムのデバッグに使用されます。**DR0, DR1, DR2, DR3, DR4, DR5, DR6, DR7**がそれにあたります。これ以外の方法でアクセスしようとすると、**GPF (General Protection Fault)** が発生し、トリプルフォールトになります。

ブレイクポイントレジスター

レジスタ**DR0, DR1, DR2, DR3**には、ブレイクポイント条件の**絶対アドレス**が格納されています。**ページング**が有効な場合、このアドレスは絶対アドレスに変換されます。これらのブレイクポイント条件は**DR7**でさらに定義されます。

デバッグコントロールレジスタ

DR7は32ビットのレジスタで、ビットパターンを使って現在のデバッグタスクを識別します。これは

- **Bit 0...7** - 4つのデバッグレジスタを有効にする (以下参照)
- **ビット8...14** - ?
- **ビット15...23** - ブレイクポイントがトリガされるタイミング。各2ビットは、1つのデバッグレジスタを表します。これは以下のいずれかになります。
 - **00** - 実行時のブレイク
 - **01** - データ書き込み時のブレイク
 - **10** - IOのリードまたはライトでブレイクします。現在、この機能をサポートしているハードウェアはありません。
 - **11** - データの読み取りまたは書き込み時のブレイク
- **ビット24~31** - ウォッチするメモリの大きさを指定します。各2ビットは、1つのデバッグレジスタを表します。これは以下のいずれかになります。
 - **00** - 1バイト **01** -
 - 2バイト **10** - 8バ
 - イト **11** - 4バイト

デバッグレジスタの有効化には2つの方法があります。これは、**Local**レベルと**Global**レベルです。異なる**タスク**を使用している場合（**ページング**の場合など）、すべての**Local debugの変更はそのタスクにのみ影響**します。プロセッサは、タスクを切り替える際に、すべての**Local**の変更を自動的にクリアします。一方、**Global** タスクはそうではありません。

上記リストのビット**0**～**7**では

- **Bit 0:** ローカルDR0レジスタの有効化
- **Bit 1:** グローバルDR0レジスタの有効化
- **Bit 2:** ローカルDR1レジスタの有効化
- **Bit 3:** グローバルDR1レジスタの有効化
- **Bit 4:** ローカルDR2レジスタの有効化
- **Bit 5:** グローバルDR2レジスタの有効化
- **Bit 6:** ローカルDR3レジスタの有効化
- **Bit 7:** グローバルDR3レジスタの有効化

デバッグステータスレジスタ

これは、エラー発生時に何が起ったかを判断するためにデバッグで使用されます。**プロセッサは、有効な例外エラーに遭遇すると、このレジスタの下位4ビットを設定し、例外ハンドラを実行**します。

警告デバッグステータスレジスタ（DR6）は決してクリアされません。プログラムを続行する場合は、必ずこのレジスタをクリアしてください。

モデル・スペシフィック・レジスタ

このレジスタは、他のプロセッサにはない、プロセッサ固有の機能を提供する特別なコントロールレジスタです。これらはシステムレベルであるため、**リング0**プログラムのみがこのレジスタにアクセスできます。

これらのレジスターは各プロセッサに固有のものであるため、実際のレジスターは変更される可能性

があります。**x86**には、このレジスタにアクセスするための**2**つの特別な命令があります。

- **RDMSR** - MSRからの読み出し
- **WRMSR** - MSRからの書き込み

このレジスタは非常にプロセッサに依存しています。このため、これらのレジスタを使用する前には、**CPUID**命令を使用するのが賢明です。

あるレジスタにアクセスするには、アクセスしたいレジスタを表す**アドレス**を命令に渡さなければなりません。

インテルは長年にわたり、マシン固有ではないいくつかの**MSR**を使用してきました。これらの**MSR**は、**x86**アーキテクチャの中では一般的なものです。

モデル・スペシフィック・レジスター（MSR）		
レジスタアドレス	レジスタ名	IA-32プロセッサ・ファミリー
0x0	ia32_ps_mc_addr	ペンティアムプロセッサ
0x1	ia32_ps_mc_type	ペンティアム 4 プロセッサ
0x6	ia32_ps_monitor_filter_size	ペンティアムプロセッサ
0x10	ia32_time_stamp_counter	ペンティアムプロセッサ
0x17	ia32_platform_id	P6 プロセッサ
0x1B	IA32_APIC_BASE	P6 プロセッサ
0x3A	ia32_feature_control	Pentium 4 / Processor 673
0x79	ia32_bios_updt_trig	P6 プロセッサ
0x8B	ia32_bios_sign_id	P6 プロセッサ
0x9B	ia32_smm_monitor_ctl	Pentium 4 / Processor 672
0xC1	IA32_PMC0	インテル Core Duo
0xC2	IA32_PMC1	インテル Core Duo
0xE7	IA32_MPERF	インテル Core Duo
0xE8	IA32_APERF	インテル Core Duo
0xFE	IA32_MTRRCAP	P6 プロセッサ
0x174	ia32_sysenter_cs	P6 プロセッサ
0x175	ia32_sysenter_esp	P6 プロセッサ
0x176	ia32_sysenter_ip	P6 プロセッサ

ここに掲載されている以外にも多くの**MSR**があります。完全なリストは、[インテル開発マニュアルの付録B](#)をご覧ください。このシリーズはまだ開発中なので、どの**MSR**を参照するかはわかりません。必要に応じてこのリストに追加していきます。

RDMSR命令

CX で指定された **MSR** を **EDX:EAX** にロードします。

この命令は**特権的な命令**であり、**リング0（カーネルレベル）**でしか実行できません。**一般保護フォルト**、または**トリプルフォルト** または、**CS**の値が有効な**MSR**アドレスを表していない場合にも発生します。この命令はどの**フラグ**にも影響しません。

ここでは、この命令を使った例を紹介します（この例は、後のチュートリアルでも出てきます）。

```

; IA32_SYSENTER_CS MSRからの読み込みです。

movcx, 0x174 ; レジスター 0x174:ia32_sysenter_cs
rdmsr      ; MSRのリード

EDX:EAXには、64ビットレジスタの下位32ビットと上位32ビットが含まれます。
www.brokenthorn.com/Resources/OSDev7.html
```


WRMSR命令

EDX:EAXに格納されている64ビットの値を、CXで指定されたMSRに書き込みます。

この命令は**特権的な命令**であり、**リング0（カーネルレベル）**でしか実行できません。**一般保護フォルト**、または**トリプルフォルト** また、**CS**の値が有効な**MSR**アドレスを表していない場合にも発生します。この命令はどの**フラグ**にも影響しません。

ここでは、その使い方の一例を紹介します。

```
IA32_SYSENTER_CS MSRに書き込みます。

movcx, 0x174 ; レジスター 0x174:ia32_sysenter_cs
wrmsr      EDX:EAXのMSRへの書き込み
```

コントロールレジスター

これは、私たちにとって重要なことになります。

コントロールレジスタは、プロセッサの動作を変更することができます。それらは**CR0**、**CR1**、**CR2**、**CR3**、**CR4**です。

CR0制御レジスタ

CR0は、プライマリ・コントロール・レジスタです。32ビットで、以下のように定義されています。

- **ビット0（PE）**：システムを**プロテクトモード**にする
- **Bit 1（MP）**：Monitor Coprocessor Flag WAIT命令の動作を制御します。ビット2（EM）：エミュレートフラグ。セットされていると、**コプロセッサ命令**で**例外が発生**します。 **Bit 3（TS）**：Task Switched
- **Flag** プロセッサが他の**タスク**に切り替わったときにセットされます。 **ビット4（ET）**：ExtensionType
- **Flag（拡張タイプフラグ）**。搭載されている**コプロセッサの種類**を教えてください。
 - 0 - 80287がインストールされている
 - 1 - 80387がインストールされています。
- **ビット5（NE）**。数値エラー
 - 0 - 標準エラー報告の有効化
 - 1 - 内部x87 FPUのエラー報告を有効にする
- **ビット6-15**：未使用
- **ビット16（WP）**。ライトプロテクト
- **ビット17**：未使用
- **ビット18（AM）**。アライメントマスク
 - 0 - アライメントチェック Disable
 - 1 - アライメントチェック有効（EFLAGSと**リング3**に**ACフラグ**が設定されていることも必要
- **ビット19～28**未使用
- **ビット29（NW）**。非ライトスルー
- **ビット30（CD）**。キャッシュディセイブル
- **Bit 31（PG）**：メモリページングを有効にします。
 - 0 - Disable
 - 1 - 有効にして**CR3レジスタ**を使用

うわあ...新しいことがいっぱいだね。ビット0は、**システムをプロテクトモード**にします。つまり、**CR0レジスタのビット0を設定**することで、**実質的にプロテクトモードに入る**ことになります。

例えば、以下のように。

```
mov ax, cr0 ; CR0の値を取得
and ax, 1   ; ビット0を設定するとプロテクトモードになる
or ax, 1    ; ビット0が設定されたので、32ビットモードになりました。
mov
```

すげえええええええそんなに簡単なんですか！？そうではありません。)

このコードをブートローダに入れると、ほぼ確実に**トリプルフォルト**になります。プロテクトモードはリアルモードとは異なる**メモリアドレッシングシステム**を使用しています。また、**pmodeには割り込みがないことを覚えておいてください**。また、**pmode**には割り込みがありませんので、**1回のタイマー割り込みでもトリプルフォルト**になります。また、異なる**アドレッシングモデル**を使用しているため、**CSは無効**です。32ビットコードにするためには、**CS**をアップデートする必要があります。また、**メモリマップに特権レベルを設定していません**。

詳細は後述します。 **CR1 Control**

Register インテルで予約されて

いるため、使用しないでください。

CR2コントロールレジスタ

ページフォルトリニアドレス。ページフォルト例外が発生した場合、**CR2**にはアクセスが試みられたアドレスが格納されます。

CR3コントロールレジスタ

CR0のPGビットがセットされている場合に使用されます。最後の20ビットはページディレクトリベースレジスタ（PDBR）を含みます。

CR4コントロールレジスタ

プロテクトモードで使用され、**v8086モード**、**I/Oブレイクポイントの有効化**、**ページサイズの拡張**、**マシンチェックの例外**などの動作を制御します。

このフラグのどれかを使うかどうかはわかりません。念のため、ここに記載しておきます。これらが何であるか理解できなくても、あまり気にしないでください。

- **Bit 0 (VME) :** Virtual 8086 Mode Extensionsを有効にする。

ズ

- **ビット1 (PVI)** : プロテクトモードの仮想インタラプトを有効にする^ズ
- **ビット2 (TSD)** : タイムスタンプイネーブル
 - 0 - RDTSC命令は、どの特権レベルでも使用可能
 - 1 - RDTSC命令は、リング0でのみ使用可能
- **Bit 3 (DE)** : デバッグ拡張機能の有効化
- **ビット4 (PSE)** : ページサイズ拡張 0
 - - ページサイズは4KB
 - 1 - ページサイズは4MB。PAEでは2MBとなります。
- **ビット5 (PAE)** : 物理アドレス拡張 **ビット6 (MCE)** : マシンチェックエクセプション **ビット7 (PGE)** : ページグローバルイネーブル
- **ビット8 (PCE)** : パフォーマンス・モニタリング・カウンタ・イネーブル
 - 0 - RDPMC命令は、どの特権レベルでも使用可能
 - 1 - RDPMC命令は、リング0でのみ使用可能
- **9ビット (OSFXSR)** : FXSAVEおよびFXSTOR命令 (SSE) に対するOSのサポート
- **10ビット (OSXMMEXCPT)** : マスクされていないSIMD FPU例外に対するOSのサポート
- **11-12ビット** : 未使用
- **ビット 13 (VMXE)** : VMX イネーブル

CR8コントロールレジスタ

タスクプライオリティレジスタ (TPR) へのリード/ライトアクセスを提供する

PMode Segmentation Registers

x86ファミリでは、複数のレジスタを使用して、各**セグメント記述子**の現在のリニアアドレスを格納しています。これについては後述します。こ

れらのレジスタは

- **GDTR** - グローバルディスクリプタテーブルレ
- ジスタ **IDTR** - インタラプトディスクリプタテ
- ーブルレジスタ **GDTR** - ローカルディスクリプ
- タテーブルレジスタ **TR** - タスクレジスタ

次のセクションでは、これらのレジスタについて詳しく見ていきます。

プロセッサアーキテクチャ

このシリーズでは、プロセッサとマイクロコントローラの間に多くの類似点があることに気付きます。つまり、マイクロコントローラはプロセッサと同じようにレジスタを持ち、命令を実行します。CPU自体は**専用のコントローラチップに過ぎません**。

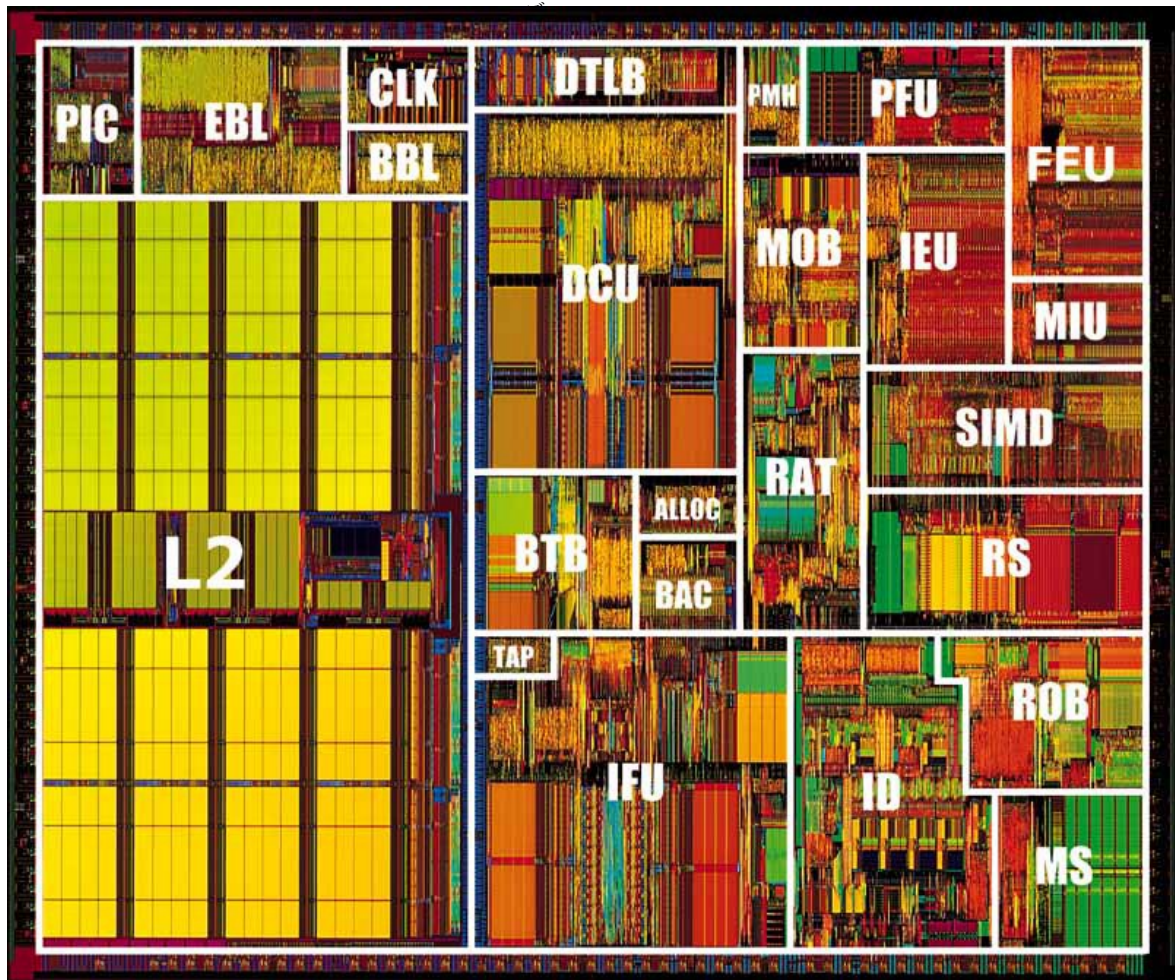
ブートプロセスについては、もう少し後に、非常に低いレベルから見ていきます。これにより、BIOSのPOSTが実際にどのように開始されるのか、またPOSTの実行、プライマリプロセッサの起動、BIOSのロードなどに関する多くの疑問が解決されます。ここまでは「**何をするか**」については説明しましたが、「**どのようにするか**」についてはまだ説明していません。

注 : このセクションはかなり専門的です。すべてを理解する必要はありませんので、すべてを理解できなくても心配はありません。私がこのセクションを設けたのは、**コンピュータシステムに必要な主要コンポーネント、そして私たちのコードを実行する責任のあるコンポーネントを完全なものにするためです**。与えられたコードをどのように実行するのか？機械語の何がそんなに特別なのか？これらの疑問にお答えします。

この後、**カーネルやデバイスドライバ**の開発に入ると、基本的なハードウェアコントローラのコンポーネント自体を理解することは、素晴らしい学習経験になるだけでなく、そのコントローラのプログラミング方法を理解するために必要なこともあることがわかります。

プロセッサを分解する

ここでは、説明のためにPentium IIIプロセッサを取り上げます。まず、このプロセッサを開いて、個々の部品に分解してみましょう。



プロセッサにはいろいろなものが入っていますね。いかに複雑であるかがわかります。この写真だけではよくわからないので、それぞれの部品を見てみましょう。

- **L2**: レベル2キャッシュ
- **CLK**: クロック
- **PIC**: プログラマブルインタラプトコントローラ **EBL**: フロントバスロジック
- **BBL**: バック・バス・ロジック
- **IEU**: 整数実行ユニット
- **FEU**: 浮動小数点演算ユニット
- **MOB**: メモリオーダーバッファ
- **MIU / MMU**: Memory Interface Unit / Memory Management Unit
- **DCU**: Data Cache Unit
- **IFU**: Instruction Fetch Unit
- **ID, ROB**: リオーダーバッファ
- **ア**
- **MS**: マイクロインストラクション・シーケンサ
- **BTB**: ブランチターゲットバッファ
- **ブ**
- **BAC**: Branch Allocator Buffer
- **RAT**: Register Alias Table
- **SIMD**: Packed Floating Point Data TLB
- **DTLB**: Data TLB
- **RS**: リザベーションステーション
- **PMH**: Page Miss Handler
- **PFU**: Pre-Fetch Unit
- **TAP**: テストアクセスポート

私はこのセクションに追加する予定です。

命令の実行方法

さて、**IP**レジスターには、現在実行中の命令のオフセットアドレスが入っていることを覚えておきましょう。**CS**はセグメントアドレスを含みます。

では、プロセッサが命令を実行する際には、具体的にどのようなことが起こるのでしょうか。

まず、読み出す必要のある絶対アドレスを計算します。**segment:offset**モデルでは、絶対アドレス = $\text{segment} * 16 + \text{offset}$ であることを覚えておいてください。あるいは、基本的には「絶対アドレス = $\text{CS} * 16 + \text{IP}$ 」となります。

プロセッサはこのアドレスをアドレスバスにコピーします。アドレスバイトは、それぞれが1つのビットを表す一連の電子線であることを覚えておいてください。このビットパターンは、次の命令の絶対アドレスのバイナリ形式を表しています。

この後、プロセッサは「Read Memory」ラインをイネーブルにします（ビットを1に設定します）。これにより、メモリコントローラに、メモリからの読み出しが必要であることを伝えます。

は、コントロールバスに設定されています。

メモリコントローラがコントロールバスをリセットすることで、プロセッサは実行の完了を認識します。プロセッサはデータバス内の値を受け取り、デジタルロジックゲートを使ってそれを「実行」します。この「値」は、一連の電子パルスとしてエンコードされた機械命令のバイナリ表現に過ぎません。

例えば、**mov ax, 0x4c00**という命令を実行した場合、**0xB8004C**という値がプロセッサのデータバス上に現れます。**0xB8004C**は**オペコード (OPCode)**と呼ばれるものです。すべての命令には、それに関連するオペコードがあります。**i86**アーキテクチャの場合、今回の命令はオペコード「**0xB8004C**」と評価されます。この数字を2進数に変換すると、電子線のようなパターンが見えてきます。**1**はその線がハイ（アクティブ）であることを意味し、**0**はその線がローであることを意味します。

```
101110000000000001001100
```

プロセッサは、**CPU**のデジタル論理回路に組み込まれた一連の離散的な命令に従います。これらの命令は、一連のビットをどのようにエンコードするかをプロセッサに指示します。すべての**x86**プロセッサは、このビットパターンを**mov ax, 0x4c00**命令としてエンコードします。

命令の複雑化に伴い、ほとんどの新しいプロセッサは独自の内部命令セットを採用しています。これはプロセッサに限ったことではなく、マイクロコントローラでも電子機器の複雑さを軽減するために複数の内部命令セットを使用している場合があります。通常、これらは**マクロコード**と**マイクロコード**です。

マクロコードとは、プロセッサが命令をマイクロコードにデコードする際に使用する抽象的な命令群のことである。マクロコードは通常、電子技術者が開発した特殊なマクロ言語で書かれ、コントローラ内の**ROM**チップに格納され、マクロアセンブラでコンパイルされる。マクロアセンブラは、マクロコードをさらに低レベルの言語（コントローラの言語）にアセンブルする。マイクロコード。

マイクロコードは、電子技術者が開発した非常に低いレベルの言語です。マイクロコードは、コントローラやプロセッサが命令をデコードするために使用されます...例えば、**0xB8004C (mov ax, 0x4c00)** 命令などです。

CPUは、**ALU (Arithmetic Logic Unit)** を使って、**0x4C00**という数字を得ることができます。そして、それを**AX**にコピーします（単純なビットコピーです）。

この例では、すべてがどのように組み合わせられるかを示しています。**CPU**がどのようにシステムバスを使用し、**メモリコントローラ**がどのように**メモリロケーション**をデコードし、**コントロールバス**に従うかを説明しています。

これは重要なコンセプトです。ソフトウェアポートも同様に**メモリーコントローラ**に依存しています。

プロテクトモード - 理論

さて、ではなぜアーキテクチャの話をしたのでしょうか？実はプロテクトモードでは、割り込みが発生しないのです。つまり...割り込みがない。システムコールもない。標準ライブラリもない。何もない。すべて自分でやらなければならないのです。このため、私たちを導いてくれる救いの手はありません。たった一つのミスでシステムがクラッシュしたり、注意を怠るとハードウェアが破壊されてしまうこともあります。フロッピーディスクだけでなく、ハードディスク、外付け（および内蔵）デバイスなど。

システム・アーキテクチャを理解することで、すべてをよりよく理解することができます、ミスをしないようにすることができます。また、私たちができる唯一のことである、ハードウェアの直接プログラミングの入門にもなります。

と思っているかもしれません。待てよ、約束していた**超1337C**カーネルはどうなるんだ！？しかし、**C**言語はある意味で低レベル言語であることを忘れてはならない。インラインアセンブリによって、ハードウェアとのインターフェースを作ることができる。また、**C**は**C++**と同じように、プロセッサが直接実行できる**x86**マシン命令のみを生成します。ただし、**標準ライブラリがないことを忘れてはいけません**。また、たとえ**高レベルの言語を使っている、非常に低いレベルの環境でプログラミング**をしていることを忘れてはいけません。

これについては、カーネルを起動するときに説明します。

結論

この種のチュートリアルを書くのは好きではありません。膨大な量の情報が詰め込まれていて、コードは少なく、理解を深めるためにコンセプトを具体的に説明しています。単純に書くのが大変なんですよ。

私はすべてを十分に説明したと思います。メモリマッピング、ポートマッピング、**x86**のポートアドレス、**x86**の全レジスタ、**x86**のメモリマップ、システムアーキテクチャ、**IN/OUT**キーワードとその実行方法、命令の実行方法などを順を追って説明しました。また、基本的なハードウェア・プログラミングについても見てきましたが、これからたくさんをやっていくことになります。

次のチュートリアルでは、**32ビットの世界へようこそ！**ということで、切り替えを行います。また、切り替えには**GDT**が必要になりますので、**GDT**についても詳しく説明します。また、よくあるエラーについては、すべてのステップで警告するつもりです。前にも言いましたが、プロテクトモードに入るときにちょっとしたミスをする、プログラムがクラッシュしてしまいます。

楽しみです... :)次の機会ま

で。

マイク
BrokenThorn Entertainment 社。現在、**DoE**と**Neptune Operating System**を開発中です。質問や

コメントはありますか？お気軽にお問い合わせください。

あなたも記事の改善に貢献したいと思いませんか？もしそうなら、ぜひ**私に教えてください**。





オペレーティングシステム開発シリーズ

オペレーティングシステム開発 - プロテクトモード

by Mike, 2008

このシリーズは、オペレーティングシステムの開発を一から実演し、教えることを目的としています。

はじめに

歓迎します。:)

このシリーズでは、これまでに多くのことを取り上げてきました。ブートローダ、システムアーキテクチャー、ファイルシステム、リアルモードアドレッシングなどを詳しく説明しました。これは素晴らしいことですが、まだ**32ビット**の世界を見ていません。そして、私たちは**32ビットのOS**を作らないのでしょうか？

このチュートリアルでは、**32ビットの世界**に飛び込んでみようと思います。もちろん、まだ**16ビット**の世界を終えたわけではありませんが、プロテクトモードに入るのは今の方がずっと簡単です。

では、早速始めてみましょう。このチュートリアル

- ではプロテクトモードの理論
- プロテクトモードアドレッシング
- プロテクトモードへの移行 グロ
- ーバルディスクリプターテーブル (GDT)

いいですか？

stdio.inc

物事をよりオブジェクト指向にするために、すべての入出力ルーチンを**stdio.inc**ファイルに移しました。くれぐれも、C言語の標準ライブラリである**stdio.lib**と一緒にしないでください。両者にはほとんど共通点がありません。カーネルの開発と並行して、標準ライブラリの開発にも着手する予定です。

とにかく...ファイルをどうぞ。

```
;*****
;      stdio.inc
;      入力/出力のルーチン
;
;      OS開発シリーズ
;*****
#ifndef STDIO_INC_67343546FDCC56AAB872_INCLUDED
#define STDIO_INC_67343546FDCC56AAB872_INCLUDED

;*****;
; Puts16 ()
; -ヌルで終端する文字列を印刷する
; DS=>SI: 0で終了する文字列
;*****;

ビット    16

Puts16。
        pusha                レジスタの保存
.Loop1:
        lodsb                SIからALへの文字列から次のバイトを読み込む
        or     al, al        AL=0の場合は？
        jz     Puts16Done    そうだ、ヌルターミネーターが見つかったから脱出しよう
        mov    ah, 0eh
        int    10h           ; Nope-文字を印刷する
        jmp    .Loop1        BIOSを起動する
Puts16Done
です。    ボバ                ; ヌルのターミネーターが見つかるまで繰り返す
        レト                レジスタの復元
                                終わったので、リターン

#endif ; STDIO_INC_67343546FDCC56AAB872_INCLUDED
```

ご存知ない方のために--*.INCファイルはインクルードファイルです。このファイルに必要に応じて追加していきます。**puts16**の説明はしませんがこの関数は、ブートローダで使ったものと全く同じルーチンで、**pusha/popa**が追加されています。

ステージ2へようこそ

ブートローダは小さい。役に立つことをするには小さすぎます。ブートローダは、**512バイト**に制限されています。それ以上でもそれ以下でもありません。真面目な話、**Stage 2**をロードするためのコードは、ほとんど**512バイト**だったのです。単純に小さすぎるのです。

2021/11/15 13:05

オペレーティングシステム開発シリーズ

これが、ブートローダが別のプログラムを*単に*ロードすることを望む理由です。FAT12ファイルシステムでは、2つ目のプログラムは、ほとんどすべてのセクタ数に対応しています。このため、512バイトの制限はありません。これは素晴らしいことです。読者の皆さん、これがステージ2です。

ステージ2のブートローダは、カーネルのためにすべての設定を行います。これは、WindowsのNTLDR (NT Loader) に似ています。実際、私はこのプログラムをKRNLDR (Kernel Loader) と名付けています。ステージ2はカーネルのロードを担当しますので、KRNLDR.SYSとしました。

KRNLDR -- 当社のStage 2ブートローダは、いくつかの機能を備えています。それは以下のことです。

- プロテクトモードの有効化と移行 **BIOS情報の取得**
- **カーネルのロードと実行**
- 事前のブートオプションの提供（例えば、セーフモードなど）
- 設定ファイルにより、KRNLDRを複数のOSカーネルから起動させることができます。
- **20番目のアドレスラインを有効にして、最大4GBのメモリにアクセス可能**
- **本的な割り込み処理を提供**

...などです。実際、Stage 2のローダーにはC言語とx86アセンブリーが混在していることが多いです。

ご想像の通り、ステージ2のブートローダを書くことは、それだけで大きなプロジェクトになります。しかし、すでに動作しているカーネルなしに、高度なブートローダを開発することはほとんど不可能です。このため、私たちは上記の**太字**で示した重要な詳細についてのみ心配するつもりです。カーネルが動作するようになったら、またブートローダの話に戻るかもしれません。

ここでは、まずプロテクトモードに入ることを考えます。皆さんは、32ビットの世界に入りたくてウズウズしているのではないのでしょうか。

プロテクトモードの世界

Yippie!いいよいですね。プロテクトモード」という言葉を何度も耳にしたことがあると思いますが、以前にも詳しく説明しました。ご存知の通り、プロテクトモードはメモリの保護を目的としています。メモリをどのように使用するかを定義することで、特定のメモリロケーションが変更されたり、コードとして実行されたりしないようにすることができます。80x86プロセッサは、**GDT (Global Descriptor Table)** に基づいてメモリ領域をマッピングしています。**GDTに従わないと、プロセッサはGPF (General Protection Fault) 例外が発生させます。割り込みハンドラを設定していないので、これはトリプルフォールトになります。**

それでは、もう少し詳しく見てみましょう。

記述子のテーブル

記述子テーブルは、何かを定義したり、マッピングしたりするもので、ここではメモリと、そのメモリがどのように使用されるかを定義しています。記述子テーブルには3種類あります。**グローバルディスクリプターテーブル (GDT)**、**ローカルディスクリプターテーブル (LDT)**、**インタラプトディスクリプターテーブル (IDT)** で、それぞれのベースアドレスは、**GDTR、LDTR、IDTRのx86プロセッサレジスタに格納されています。**これらは特殊なレジスタを使用するため、特殊な命令が必要となります。**注意：これらの命令の中には、Ring 0カーネルレベルのプログラムに特有のものががあります。一般のRing 3プログラムがこれらを使おうとすると、GPF (General Protection Fault) 例外が発生します。今回のケースでは、まだ割り込みを処理していないので、トリプルフォールトが発生します。**

グローバル記述子テーブル

これは私たちにとって重要なことであり、ブートローダとカーネルの両方で見ることができます。

グローバルディスクリプターテーブル (GDT) は、グローバルなメモリーマップを定義しています。どのメモリが実行できるか (**コード記述子**)、どの領域にデータがあるか (**データ記述子**) を定義している。

記述子はプロパティを定義するもの、つまり何かを記述するものであることを覚えておいてください。GDTの場合、記述子には開始アドレス、ベースアドレス、セグメントの制限、さらには仮想メモリが記述されています。これは実際に見てみるとよくわかると思いますが、ご心配なく。)

GDTは通常、**Null記述子** (すべてのゼロを含む)、**コード記述子**、**データ記述子**の3つの記述子を持っています。なるほど では、「記述子」とは何で

しょうか？GDTでは、ディスクリプターのプロパティを記述した8バイトのQWORD値を指します。形式になっています。

- **56~63ビット** : ベースアドレスの24~32ビット目
- **ビット55** : グラニュラリティ
 - **0** : なし
 - **1** : リミットが4K倍になります。
- **ビット54** : セグメントタイプ
 - **0** : 16ビット
 - **1** : 32ビット
- **ビット53** : 予約済み-0にすべき
- **ビット52** : OS用の予約
- **ビット48~51** : セグメントリミットのビット16~19
- **ビット47** : セグメントがメモリ内にある (仮想メモリで使用)
- **ビット45-46** : ディスクリプターの特権レベル
 - **0** : (Ring 0) Highest
 - **3** : (Ring 3) Lowest
- **ビット44** : ディスクリプタービット
 - **0** : システム記述子
 - **1** : コードまたはデータ記述子
- **ビット41-43** : ディスクリプター・タイプ
 - **ビット43** : 実行可能セグメント
 - **0** : データセグメント
 - **1** : コードセグメント
 - **ビット42** : 拡張方向 (データセグメント)、準拠 (コードセグメント)
 - **ビット41** : 読み出し可能、書き込み可能
 - **0** : 読み出しのみ (データセグメント)、実行のみ (コードセグメント)
 - **1** : 読み取りと書き込み (データセグメント)、読み取りと実行 (コードセグメント)
- **ビット40** : アクセスビット (仮想記憶で使用) **ビット16~39** : ベースアドレスの0-23
- **ビット0-15** : セグメントリミットの0-15ビット

2021/11/15 13:05

オペレーティングシステム開発シリーズ

...かなり醜いですね？基本的には、ビットパターンを構築することで8バイトのビットパターンがディスクリプターのさまざまなプロパティを記述することになる。各ディスクリプターは、そのメモリセグメントのプロパティを定義します。

簡単に説明すると、メモリの第1バイトから第0xFFFFFFFFバイトまでの読み書き可能なコードとデータの記述子を定義したテーブルを作ります。これは、メモリ上の任意の場所を読み書きできることを意味します。

まず、GDTを見てみましょう。

```

; これはGDTの始まりです。このため、そのオフセットは0です。

ヌルデスクリプター
    dd 0                      ; ヌルデスクリプター--8バイトをゼロで埋めるだけ
    dd 0

; 各デスクリプターのサイズが正確に8バイトであることに注目してください。これは重要なことです。
; このため、コード記述子のオフセットは0x8となっています。
; コード記述子。
    dw 0FFFFh                ; コード記述子。ヌルヌル記述子の直後
    dw 0                      限界の低さ
    db 0                      ベースの低さ
    db 0                      ベース、ミドル
    db 10011010b              アクセス
    db 11001111b              グラニューラリティ
    db 0                      基地の高さ

各記述子のサイズは8バイトなので、データ記述子は、オフセット0x10で
GDTの先頭、または先頭から16 (10進数) バイトのところ。

データ記述子。
    dw 0FFFFh                ; データ記述子
    dw 0                      コードと同じ
    db 0                      ベースの低さ
    db 0                      ベース、ミドル
    db 10010010b              アクセス
    db 11001111b              グラニューラリティ
    db 0                      基地の高さ

```

それはそれは。悪名高きGDTです。このGDTには3つの記述子があり、それぞれ8バイトの大きさです。ヌルの記述子、コード、データの記述子である。各記述子の各ビットは、上記のビットテーブル（コードの上に表示）で表されるビットに直接対応している。

何が起きているのか、それぞれのビットに分解してみましょう。ヌルデスクリプターはすべてゼロなので、残りの2つに注目します。

コードセクターを分解する

もう一度見てみましょう。

```

; コード記述子。
    dw 0FFFFh
    dw 0
    db 0
    db 10011010b
    db 11001111b
    db 0

; コード記述子。ヌルヌル記述子の直後
    限界の低さ
    ベースの低さ
    ベース、ミドル
    アクセス
    グラニューラリティ
    基地の高さ

```

アセンブリ言語では、宣言された各バイト、ワード、ワード、qワード、命令などは、文字通り直後にあることを覚えておいてください。上の例では、0xffff はもちろん、1で埋め尽くされた2バイトです。これを簡単にバイナリ形式に分解できるのは、そのほとんどがすでに行われているからです。

```
11111111 11111111 00000000 00000000 00000000 10011010 11001111 00000000
```

上のビットテーブルで、**0-15ビット（最初の2バイト）**がセグメントの制限を表していることを覚えておいてください。これはつまり、0xffff（最初の2バイト）より大きいアドレスをセグメント内で使用することはできないということです。そうするとGPFが発生します。

16-39ビット（次の3バイト）は、ベースアドレス（セグメントの開始アドレス）の0-23ビットを表しています。今回の例では0x0です。ベースアドレスが0x0で、リミットアドレスが0xFFFFなので、コードセクターは0x0から0xFFFFまでのすべてのバイトにアクセスできることになる。いいでしょ？

次のバイト（バイト6）では、面白いことが起こります。文字通り、少しずつ説明していきましょう。

```
db 10011010b          アクセス
```

- **ビット0（GDTではビット40）**。アクセスビット（仮想メモリで使用）。仮想メモリを使用しないので（まだ）、無視します。そのため、0です。
- **第1ビット（GDTでは第41ビット）**：読み取り/書き込み可能なビットです。コードセクターにセットされているので、0x0から0xFFFFまでのセグメントのデータをコードとして読み出し、実行することができます。
- **第2ビット（GDTでは第42ビット）**：「拡張方向」のビットです。これについては後ほど詳しく見ていきます。今のところは無視してください。
- **第3ビット（GDTでは第43ビット）**：これがコードまたはデータの記述子であることをプロセッサに伝える。（セットされているので、コード記述子であることを示しています。）
- **ビット4（GDTではビット44）**。システムまたは「コード/データ」の記述子として表します。これはコードセクターなので、このビットは1に設定されます。
- **ビット5-6（GDTではビット45-46）**：特権レベル（リング0かリング3か）です。今回はRing 0なので、両ビットとも0です。
- **ビット7（GDTではビット47）**。セグメントがメモリ内にあることを示すために使用されます（仮想メモリで使用）。まだ仮想メモリを使用していないので、今は0に設定する

アクセスバイトは非常に重要です。Ring 3のアプリケーションやソフトウェアを実行するためには、さまざまな記述子を定義する必要があります。この点については、カーネルに入ってからもっと詳しく見ていきます。

これをまとめると、このバイトは示しています。これは読み書き可能なセグメントであり、我々はリング0のコード記述子である。

次のバイトを見てみましょう。

```
db 11001111b      グラニュラリティ
db 0               基地の高さ
```

グラニュラリティバイトを見て、それを分解してみましょう。上記のGDTビットテーブルをご参照ください。

- **ビット0～3 (GDTではビット48～51)**。セグメントリミットのビット16～19を表します。つまり、レ1111bは0xfと同じです。このディスクリプターの最初の2バイトでは、最初の15ビットに0xffffを設定していることを思い出してください。低位ビットと上位ビットをグループ化すると、**0xFFFFFまでアクセスできるようになります**。いいですか？もっといいことがある...。20番目のアドレスラインを有効にすることで、このディスクリプターを使って**最大4GBのメモリ**にアクセスすることができます。これについては後で詳しく見てみましょう...。
- **ビット4 (GDTではビット52)**。我々のOSの使用のために予約されており、ここで何でもできる。0に設定されています。
- **ビット5 (GDTではビット53)**。何かのために予約されています。将来のオプションかな？分からない。0に設定してください。
- **ビット6 (GDTではビット54)**：セグメントタイプ (16ビットまたは32ビット) です。レスです。私たちは32ビットを望んでいますよね。結局のところ、私たちは32ビットのOS! そうそう ---- 1に設定します。
- **ビット7 (GDTではビット55)**。グラニュラリティ。1に設定すると、各セグメントの境界は4KBになります。

最後のバイトは、ベース (スタート) アドレスのビット24～32です。 はもちろん0です

。

それだけでいいんだよ！」。

データ記述子

それでは、作成したGDTに戻って、コードセクターとデータセクターを比較してみましょう。両者は、**1つのビットを除いてまったく同じです。43ビットです**。上の図を振り返ると、その理由がわかります。コードセクターの場合はセットされ、データセクターの場合はセットされません。

結論

これまでに見た (書いた) GDTの説明の中で、最も包括的なものです！でも、それは良いことですよね？

はいはい、GDTは醜いですがからね。しかし、それをロードして使用するのは非常に簡単です。実際には、ポインターのアドレスをロードするだけでいいのです。

このGDTポインタには、GDTのサイズ (**マイナス1!**) とGDTの先頭アドレスが格納されています。例えば、以下のようになります。

```
gdt_dataはGDTの先頭、end_of_gdtはもちろんGDTの末尾のラベルです。このポインタの大きさと、そのフォーマットに注意してください。
GDTのポインタはこのフォーマットに従わなければなりません。それがないと、予測できない結果になります。最も可能性が高いのはトリプルフォールトです。
GDT) dd gdt_data; base of GDT
```

プロセッサは、ベースとなるGDTポインタ内のデータを格納する特別なレジスタ--**GDTR**を使用します。GDTをGDTRレジスタにロードするためには、特別な命令が必要です...。 **LGDT** (Load GDT) です。使い方はとても簡単です。

```
lgdt[          GDTのGDTRへのロード
toc] (
```

これは冗談ではなく、本当に簡単なことなのです。今回のように、OSデヴェューのような素敵なブレイクはそうそうありません。今のうちに確保しておきましょう。

ローカルディスクリプターテーブル

ローカルディスクリプターテーブル (LDT) は、特殊な用途のために定義されたGDTの縮小版である。システムのメモリマップ全体を定義するのではなく、最大で**8,191個**のメモリセグメントのみを定義します。これはプロテクトモードとは関係ないので、後で詳しく説明します。いいですか？

割り込みディスクリプターテーブル

これが重要になります。まだまだですが、割り込みディスクリプターテーブル (IDT) は、割り込みベクターテーブル (IVT) を定義します。最初の32個のベクターは、プロセッサが生成するハードウェア例外のために予約されています。例えば、**一般保護フォールト**や**ダブルフォールト例外**などです。これにより、トリプルフォールトを起こさずにプロセッサのエラーをトラップすることができます。これについては、後で詳しく説明します。

その他の割り込みベクターは、マザーボード上の**Programmable Interrupt Controller**チップを通じてマッピングされます。プロテクトモードでは、このチップを直接プログラムする必要があります。これについては後で詳しく説明します。

PMode メモリアドレッシング

PMode(Protected Mode)はリアルモードとは異なるアドレッシングスキームを使用していることを覚えておいてください。リアルモードでは、**セグメント : オフセット**

しかし、PModeは**Descriptor:Offset**モデルを採用しています。

つまり、PModeでメモリにアクセスするためには、GDT内の正しいディスクリプターを経由する必要があります。このディスクリプターはCSに格納されています。これにより、現在のディスクリプター内のメモリを間接的に参照することができます。

例えば、あるメモリロケーションから読み出す必要がある場合、どのディスクリプターを使用するかを記述する必要はなく、現在CSにあるディスクリプターを使用します。つまり、これでうまくいくのです。

```
mov bx, byte [0x1000].
```

これは素晴らしいことですが、時には特定の記述子を参照する必要がある^ズがあります。例えば、Real ModeではGDTを使用しませんが、PModeではGDTが必要です。このため、プロテクトモードに入る際には、プロテクトモードでの実行を継続するために**どのディスクリプターを使用するかを選択する必要があります**。結局のところ、Real ModeはGDTが何であるかを知らないで、CSに正しい記述子が含まれているという保証はないので、それを設定する必要があるのです。

そのためには、ディスクリプタを直接設定する必要があります。

```
jmp     0x8:Stage2
```

このコードをもう一度見てみましょう。最初の数字が**記述子**であることを覚えておいてください（PModeが記述子：アドレスのメモリモデルを使っていることを覚えていますか？）

0x8がどこから来たのか、不思議に思うかもしれません。上のGDTを見てください。**各記述子のサイズは8バイトであることを覚えておいてください**。**Code**の記述子はGDTの先頭から8バイト離れているので、GDTの中で0x8バイトオフセットする必要があります。

このメモリモデルを理解することは、プロテクトモードの仕組みを理解する上で非常に重要です。

プロテクトモードへの移行

プロテクトモードに入るのはとても簡単です。その一方で、非常に手間のかかる作業でもあります。プロテクトモードに入るためには、メモリにアクセスする際のパーミッションレベルを記述した新しいGDTをロードする必要があります。そして、実際にプロセッサをプロテクトモードに切り替えて、32ビットの世界に飛び込む必要があるのです。簡単なことだと思いませんか？

問題は細部にあります。ちょっとしたミスでCPUが3重に故障してしまう。つまり、気をつけろ！ということです。

Step 1: グローバル記述子テーブルのロード

GDTは、メモリへのアクセス方法を記述するものであることを覚えておいてください。GDTを設定しない場合、デフォルトのGDTが使用されます（これはBIOSによって設定されますが、ROM BIOSではありません）。想像できるように、これはBIOS間の標準ではありません。また、**GDTの制限に気をつけないと（例えば、コードセクタにデータとしてアクセスすると）、プロセッサはGPF（General Protection Fault）を生成します**。また、割り込みハンドラが設定されていないため、プロセッサは**第2障害例外**を発生させ、トリプルフォールトとなります。

とにかく...基本的にはテーブルを作ればいいのです。例えば

```
; GDTのオフセット0: 記述子コード=0

gdt_dataです。
    dd                                0; スルディスクリプタ
    dd 0

; GDTの開始から0x8バイトのオフセット: そのための記述子コードは8です。
gdtコード
    dw 0FFFFh                        コード記述子
    dw 0                            限界の低さ
    db 0                            ベースの低さ
    db 10011010b                     ベース、ミドル
    db 11001111b                     アクセス
    db 0                            グラニュラリティ
    db 0                            基地の高さ

; GDTの開始から16バイト (0x10) のオフセット。このため、記述子コードは0x10となります。

gdtデータ。
    dw 0FFFFh                        データ記述子
    dw 0                            コードと同じ
    db 0                            ベースの低さ
    db 10010010b                     ベース、ミドル
    db 11001111b                     アクセス
    db 0                            グラニュラリティ
    db 0                            基地の高さ

; ...その他の記述子は、オフセット0x18から始まります。各ディスクリプターのサイズが8バイトであることを覚えていますか？
; Ring 3のアプリケーションやスタックなど、他の記述子をここに追加してください...

end_of_gdtです。
toc:
    dw end_of_gdt - gdt_data -      1; limit (Size of
    gdt_data; base of GDT
```

今はこれでいいでしょう。**toc**に注目してください。これはテーブルへのポインタです。ポインタの最初のワードは、GDTのサイズ-1です。2番目のワードは、GDTの実際のアドレスです。**このポインタはこのフォーマットに従わなければなりません。1を差し引くことを忘れないでください!**

このポインタを元にしたGDTを**GDTR**レジスタにロードするには、リング0専用の**LGDT**という特別な命令を使います。これは1行のシンプルな命令です。

```
クライ          ただし、その前に必ず割り込みを解除してください
アント
lgdt    [toc]    。
sti          GDTのGDTRへのロード
```

それはそれは。簡単でしょう？さて...保護モードに入ります!うーん...そうだな!Gdt.incでGDTの醜い部分を隠しています。

```

; *****
;      Gdt.inc
;      -GDTルーチン
;
;      OS開発シリーズ
; *****

#ifdef GDT_INC_67343546FDCC56AAB872_INCLUDED
#define GDT_INC_67343546FDCC56AAB872_INCLUDED

ビット    16

; *****
; InstallGDT()
GDTのインストール
; *****
; クラ      割り込みの解除
;   イア
InstallGDTです。
; プーシ      レジスタの保存
;   ヤ
lgdt    [toc]      GDTのGDTRへのロード
;   sti      割り込みの有効化
;   ボバ      レジスタの復元
;   レッ      All done!
;   ト

; *****
グローバルディスクリプターテーブル (GDT)
; *****

gdt_dataです
。      スルデスクリプタ
      dd 0
      dd 0
gdtコード      コード記述子
      dw 0FFFFh      限界の低さ
      dw 0      ベースの低さ
      db 0      ベース、ミドル
      db 10011010b      アクセス
      db 11001111b      グラニューラリティ
      db 0      基地の高さ
gdtデータ。      データ記述子
      dw 0FFFFh      リミット下限値 (コードと同じ)
      dw 0      ベースの低さ
      db 0      ベース、ミドル
      db 10010010b      アクセス
      db 11001111b      グラニューラリティ
      db 0      基地の高さ
end_of_gdtです。
toc:
      dw end_of_gdt - gdt_data - 1; limit (Size of
      GDT) dd      gdt_data; base of GDT

#endif ; GDT_INC_67343546FDCC56AAB872_INCLUDED

```

Step 2: プロテクトモードへの移行

CR0レジスタのビットテーブルを覚えていますか？何でしたっけ？そうそう...

- **ビット0 (PE)** : システムをプロテクトモードにする
- **Bit 1 (MP) : Monitor Coprocessor Flag** WAIT命令の動作を制御します。ビット2 (EM) : エミュレートフラグ。セットされていると、**コプロセッサ命令で例外が発生します**。 **Bit 3 (TS) : Task Switched Flag** プロセッサが他のタスクに切り替わったときにセットされます。ビット4 (ET) : **ExtensionType**
- **Flag (拡張タイプフラグ)**。搭載されているコプロセッサの種類を教えてください。
 - 0 - 80287がインストールされている
 - 1 - 80387がインストールされています。
- **Bit 5** : 未使用。
- **Bit 6 (PG)** : メモリページングを有効にします。

重要なビットはビット0です。ビット0を設定することで、プロセッサは32ビットの状態で行を実行します。つまり、ビット0を設定するとプロテクトモードが有効になります。

ここではその一例をご紹介します。

```

movable    eax, cr0      CR0のビット0をセットしてpmodeへ移行
または
movable    eax, 1
movable    cr0, eax

```

2021/11/15 13:05

オペレーティングシステム開発シリーズ

これで終わりです。ビット0がセットされていれば、Bochs Emulatorはプロテクトモード(PMode)であることがわかります。

覚えておいてください。**32ビット**を指定するまでは、コードは**16ビット**のままです。コードが**16ビット**である限り、**segment:offset**メモリモデルを使用することができます。

警告!32ビットのコードに入る前に、割り込みが**DISABLED**になっていることを確認してください。これが有効になっていると、プロセッサはトリプルフォールトになります。(pmodeからIVTにアクセスできないことを覚えていますか?)

プロテクトモードに入ると、すぐに問題が発生します。リアルモードでは、**Segment:Offset**を使っていたことを思い出してください。のメモリモデルを使用しています。しかし、**Protected ModeはDescriptor:Address**のメモリモデルに依存しています。

また、リアルモードではGDTを知りませんが、PModeではアドレッシングモードのため、GDTの使用が**必須**であることを覚えておいてください。このため、リアルモードでは、CSには**使用するディスクリプターではなく**、最後に使用したセグメントアドレスが格納されています。

PModeはCSを使って現在のコード記述子を保存していることを覚えていますか?つまり、CSを修正する(コード記述子に設定する)ためには、コード記述子を使って**ファージャンプ**する必要があります。

コード記述子が**0x8** (GDTの先頭から8バイトオフセット) なので、次のようにジャンプします。

```
jmp    08h :      CSを固定するためのファージャンプ。コードセクタが0x8であることを覚えておいてください!
      Stage3
```

また、PModeに入ると、すべてのセグメント(正しくないので)を正しいディスクリプター番号にリセットする必要があります。

```
mov     ax, 0x10      データセクタ (0x10) へのデータセグメントの設定
mov     ds, ax
mov     ss, ax
mov     es, ax
abl     e
```

データ記述子がGDTの先頭から**16 (0x10)** バイトだったことを覚えていますか?

なぜ、GDT(記述子を選択する)内の参照がすべてオフセットなのか、不思議に思うかもしれません。何のオフセット? **LGDT**命令で読み込んだGDTポインタを覚えていますか?プロセッサはすべてのオフセットアドレスを、GDTポインタが指すように設定したベースアドレスを基準にしています。

Stage 2のブートローダの全体像をご紹介します。

```
ビット    16

0x500~0x7bfffは、BIOSデータエリアの上では使用されていません。
0x500 (0x50:0)でロードされます。

org 0x500
jmp     メイン          スタートまでの流れ
;*****
;      プリプロセッサ用ディレクティブ
;*****

#include "stdio.inc"      基本的なI/Oルーチン
#include "Gdt.inc"        Gdtのルーチン
;*****
;      データセクション
;*****

LoadingMsg db "Preparing to load operating system...", 0x0D, 0x0A, 0x00
;*****
;      ステージ2のエントリーポイント
;      インタ
;      -BIOS情報の保存
;      -Load Kernel
;      GDTのインストール、プロテクトモード (pmode) への移行
;      -ステージ3へのジャンプ
;*****

main:

;-----;
;      セグメントと          スタックを
;      設定します。
;-----;
;      クライアント          ; 割り込みの解除
xor     ax, ax            ; nulセグメント
movab   ds, ax
movab   es, ax
le      movab   ax, 0x9000      ; スタックは0x9000-0xffffで始まる
le      movab   ss, ax
;-----;
movab   esp, 0xffff
le      sti     メッセージ      ; 割り込みの有効化
;      を表示します
;      。
;-----;
```



```

私たちのGDT をイ ;
ンストールする-----;

        コールインス      私たちのGDTをインス
        トールGDT          トールする
;-----;

        pmodeに          割り込みの解除
        クライアント。
        movab  eax, cr0-----;cr0のビット0を設定し、pmodeに入る。
        le
        または  eax, 1
        movab  cr0, eax
        le
        jmp    08h : Stage3      CSを固定するためのファージャンプ。コードセクタが0x8であることを覚えてお
                                   いてください!

        注意： 割り込みを再度有効にしないでください。再起動すると3重に故障します
        これはStage 3で修正する予定です。

;*****
ステージ3の エントリーポイント
;*****

        ビット          32ビットの世界へようこそ。
        32
        Stage3
        です。

        ;-----;
        レジスター
        の設定
        ;-----;
        movable      ax, 0x10          データセクタ (0x10) へのデータセグメントの設定
        movable      ds, ax
        movable      ss, ax
        movable      es, ax
        movable      esp, 90000h      スタックは90000hから始まる

;*****
実行 停止
;*****

STOP:

        ク
        ラ
        イ
        ス
        ト
        チ

```

結論

私は興奮していますか、あなたはどうか？このチュートリアルでは多くのことを学びました。GDT、ディスクリプターテーブル、プロテクトモードについても説明しました。

32ビットの世界へようこそ

これは私たちにとって素晴らしいことです。ほとんどのコンパイラは32ビットのコードしか生成しないので、プロテクトモードが必要なのだ。これで、C言語でもアセンブリ言語でも、ほとんどの言語で書かれた32ビットプログラムを実行できるようになります。

しかし、まだ16ビットの世界では終わっていません。次のチュートリアルでは、BIOS情報を取得し、FAT12経由でカーネルをロードします。これはもちろん、小さな小さなスタブカーネルを作ることを意味します。かついいでしょう？

ご来場お待ちしております。

次の機会まで。

マイク

BrokenThorn Entertainment社。現在、DoEとNeptuneOperating Systemを開発中です。質問や

コメントはありますか？お気軽にお問い合わせください。

あなたも記事の改善に貢献したいと思いませんか？もしそうなら、ぜひ私に教えてください。



第7章

ホーム

第9





オペレーティングシステム開発シリーズ

オペレーティングシステムの開発 - A20を実現する by Mike, 2008

このシリーズは、オペレーティングシステムの開発を一から実演し、教えることを目的としています。

はじめに

歓迎します。:)

前回のチュートリアルでは、プロセッサを32ビットモードに切り替える方法をご紹介しました。また、最大4GBのメモリにアクセスする方法も学びました。これは素晴らしいことですが、ではどうやって？

また、PCはリアルモードで起動しますが、これには16ビットのレジスタという制限があることを覚えておいてください。また、PCはリアルモードで起動しますが、リアルモードでは16ビットのレジスタに制限があり、16ビットのセグメントアドレッシングになります。このため、リアルモードでアクセスできるメモリの量は限られています。そのため、まだ1GBのメモリにもアクセスできません。それどころか、1MBの壁も越えられません。ではどうすればいいのか。20番目のアドレスラインを有効にしなければなりません。そのためには、ハードウェアで直接プログラミングする必要がありますので、今回はその方法についてもご紹介します。

というわけで、これがメニューです。

- ダイレクト・ハードウェア・プログラミング - 理論
- ダイレクト・ハードウェア・プログラミングとコン
- トローラ キーボード・コントローラ・プログラミン
- グ - 基礎編 A20を可能にする
- ピザ :)

C言語のような高級言語を使用する場合、1MB以上のメモリにアクセスできることが重要になります。このため、A20（アドレスライン20）を有効にすることが重要になります。

注意：まだ1MB以上のアクセスはできないことを覚えておいてください。アクセスするとトリプルフォールトになります。

また、今回のチュートリアルでは、ハードウェアの直接プログラミングについて説明するため、これまでのチュートリアルよりも少し複雑になります。後日、カーネル用のデバイス・ドライバを開発する際に、ダイレクト・ハードウェア・プログラミングの経験を積むことができますので、ご安心ください。

いいですか？

準備

ここまでお付き合いいただいた方は、OS開発の大変さをご存知だと思います。しかし、ここではそれに近いものには触れていません。ここに挙げた概念はすべて、まだ基本的なものであり、かつ高度なものです。しかし。これからはもっともっと難しくなるでしょう。

すべてのコントローラーが正しく動作するためには、特別な方法でプログラムされていなければならない。例えば、ハードディスクの書き込み（読み込み）を行うには、まず、そのハードディスクがIDEドライブかSCSIドライブかを判断しなければならない。次に、そのドライブの番号を決定し、IDEとSCSIの接続を制御するIDEコントローラーまたはSCSIコントローラーを使用してプログラムしなければなりません。これらのコントローラーはそれぞれ異なります。

さらに複雑なことに、「セクタ」は512バイトではないかもしれません。そのため、「セクタの読み書き」は曖昧です。

続いて、メモリ管理とフラグメンテーション。ここでは、ページング、仮想アドレス空間、そしてMMU（Memory Management Unit）が登場します。

ドライブの読み書きは、他のドライブとは大きく異なります。これはブートセクタのレベルでも同じです。典型的なフォーマットとファイルシステムはメディアによって異なり、FAT12フロッピーから起動するコードは、CDFSファイルシステムのCD ROMを起動するためには動作しません。ハードウェア固有のコード（および低レベルのコード）を抽象化することで、ほとんどのコードをこれらのデバイス用に動作させることができます。

ハードドライブにファイルを書き込む」と言ったとき、通常、「ファイル」とは何かを定義したくはありません。なぜなら、「ファイル」とは何かを定義すべきではないからです。どのコントローラから読み取るか、ディスク上の正確な位置を気にする必要はありません。これが、抽象化が非常に重要な理由です。

ここでは、主にプロテクトモード（32ビットコード）に対応していますが、リアルモードでも動作します。このため、プロテクトモードのルールを覚えておいてください。

- 割り込みはできません。割り込みを使用すると、3重の障害が発生します。

...だから、あなたは完全に自分のものになっています。

カーネルのデバッグ

デバッグは芸術の一種です。デバッグとは、問題が深刻化する前に、問題を捕捉し、ソフトウェアのエラーを修正する方法です。

カーネルデバッグとは、カーネルレベルのRing 0プログラムのデバッグに関するものです。これは決して簡単な作業ではありません。

高レベル言語のデバッガ

CやC++などの言語のデバッガの多くは、実行時に可変長やルーチン名、その値や位置などを表示する方法を提供しています。問題は？私たちのプログラムには、まだシンボリックネームがありません。私たちはまだBinaryレベルで作業しています。

そのためには、メモリを直接操作・表示できるデバッガが必要になります。ボックスは、そんな私たちのためにデバッガを用意しました。

Bochsデバッガ

Bochsには**bochsdbg.exe**というデバッガが付属しています。これを起動すると、Bochs.exeと同じ起動画面が表示されます。設定ファイルを読み込んで、エミュレーションを開始してください。

Bochsデバッガと表示ウィンドウが表示され、行が表示されるはずですが、

```
[0x000ffff0] f000:ffff (unk. ctxt): jmp f000:e05b          ea5be000f0
< bochs:1> _。
```

2行目では、bochsが送られてきたコマンドの数を教えてくれます（この場合、最初のコマンドなので、1が表示されます）。ここでは、コマンドを入力することができます。

1行目が重要な行です。現在の命令、絶対アドレス、seg:offsetアドレスを教えてくれます。また、機械語のオペレーションコード（Opcode）に相当するものもわかります。

HELPコマンド

helpコマンドは、利用可能なコマンドの一覧を表示します。

BREAKコマンド

b (break) コマンドは、メモリ上のアドレスにブレークポイントを設定することができます。例えば、OSをデバッグしようとした場合、ブートローダ（0x7c00:0）からスタートする必要があります。しかし、Bochs Debuggerは、BIOSがあるところからスタートします。このため、0x7c00:0にブレークポイントを設定し、そのブレークポイントに到達するまで実行を継続する必要があります。

```
// BIOS is at 0xea5be000f0
[0x000ffff0] f000:ffff (unk. ctxt): jmp f000:e05b
< bochs:1>                                b 0x7c00//ブレークポイントを0x7c00:0に設定します。
< bochs:2> c// Continue execution
< 0> ブレークポイント 1, 0x7c00 in          ?? < > // ブレークポイント
がヒットしました 次は t=834339 です。
// ブートローダの最初の命令を実行します。
< 0> [0x00007c00] 0000:7c00 (unk. ctxt): jmp 7cb5; e9b200/DIV> です。
```

以上のことから、ブートローダのmain()関数は0x7cb5にあることがわかります。OEMパラメータブロックは、このジャンプ命令とmain()の開始点の間にあることを考えると、これは理にかなっています。

ブートローダがステージ2を0x500にロードすることがわかったので、それを実行します。

```
< bochs:3> b 0x500
< bochs:4> c
< 0> ブレークポイント2, 0x500 in ?? <
> 次はt=934606で。
<0> [0x000000500] 0050:0000 (ink. ctxt): jmp 00a0          e99d00
< bochs:5> _。
```

これで、ステージ2の始まりで、アセンブリファイルでデバッガをフォローできるようになりました。かっいいいでしょう？何よりも、ウィンドウが動的に更新されて、システムの出力を表示しているのがわかります。

シングルステップ

s (シングルステップ) コマンドは、1つの命令を一度に実行するために使用します。

```
< bochs:6> s
Next at t=934607
<0> [0x0000005a0] 0050:00a0 (ink. ctxt): cli          のよ
< bochs:7> s                                         うに
Next at t=934608                                     なり
<0> [0x0000005a1] 0050:00a1 (ink. ctxt): xor AX, AX   ます
< bochs:8> _。                                       。
```

dump_cpu

このコマンドは、RFLAGS、汎用レジスタ、テストレジスタ、デバッグレジスタ、コントロールレジスタ、セグメントレジスタを含む、すべてのcpuレジスタの現在の値を表示します。また、GDTR、IDTR、LDTR、TR、EIPも含まれます。

print_stack

スタックの現在の値を表示します。スタックを頻繁に使用することを考えると、これは非常に重要です。

結論

これ以外にも様々なコマンドがありますが、最も便利なのはこれらのコマンドです。デバッガーの使い方を学ぶことは、特に今のような初期段階では非常に重要です。

ダイレクト・ハードウェア・プログラミング - 理論

ここからが、OS開発の大変なところですよ。

"Direct Hardware Programming" とは、簡単に言えば、個々のチップを直接通信して（制御して）使うことです。これらのチップが（何らかの方法で）プログラム可能である限り、私たちはそれらを制御することができます。

チュートリアル7では、システムがどのように動作するかを非常に詳しく説明しました。また、ソフトウェア・ポートがどのように動作するか、ポート・マッピング、IN命令とOUT命令についても説明し、x86アーキテクチャの一般的なポート・マッピングの膨大な表を示しました。

プロセッサがIN/OUT命令を受け取ると、コントロールバスのI/Oアクセスラインが有効になることを覚えておこう。システムバスは、メモリーコントローラーとI/Oコントローラーの両方に接続されているため、両コントローラーは、コントロールバスの特定のアドレスと有効なラインを探します。I/Oアクセスラインがセットされていれば（電気が通っている＝アクティブ（1））、I/Oコントローラーはそのアドレスを受け取る。

I/Oコントローラーは、ポートアドレスを他のすべてのデバイスに与え、コントローラーチップからの信号を待ちます（あるデバイスに属していることを意味するので、そのデバイスに任意のデータを与えてください）。コントローラーチップからの応答がなく、ポートアドレスが戻されると、それは無視される。

これがポートマッピングの仕組みです。（詳しくはチュートリアル7をご覧ください）。

また、1つのコントローラーチップに複数のポートアドレスが割り当てられている場合もありますのでご注意ください。ポートアドレスは、BIOSがロードされて実行される前から、BIOSのPOSTによって割り当てられます。なぜか？多くのデバイスは異なる種類の情報を必要とします。あるポートは"レジスタ"を表し、他のポートは"データ"や"レディ"を表すかもしれません。醜いのはわかっています。しかし、さらに悪いことがあります。異なるシステムでは、ポートのアドレスは大きく異なります。x86アーキテクチャはバックワード互換なので、基本的なデバイス（キーボードやマウスなど）は通常、常に同じアドレスになります。しかし、より複雑なデバイスはそうではないかもしれません。

ダイレクトハードウェアプログラミングとコントローラー

これらの仕組みを理解するために、コントローラーを見てみましょう。特にプロテクトモードでは、コントローラーと会話することが多くなります。

多くのPCは、初期のインテル8042マイクロコントローラーチップをベースにしています。このコントローラーチップは、IC（集積回路）チップとして組み込まれているか、あるいはマザーボードに直接組み込まれている。通常はサウスブリッジに搭載されています。

このマイコンは、キーボードに接続されたコードを介して、キーボード内の別のマイコンチップと通信します。

キーボードのキーを押すと、キーの下に設置されたラバードームを押し下げます。ラバードームの下側には導電性の接点があり、押し下げられるとキーボード回路上の2つの導電性接点と接触します。これにより、電流が流れます。各鍵盤は1対の電気ラインで接続されている。それぞれの信号が変化すると（キーが押されたときに）、（一連の線から）メイクコードが生成されます。このメイクコードは、キーボード内部のマイクロコントローラーチップに送られ、コンピュータのハードウェアポートに接続するコードを介して送信されます。このコードは、オンとオフを繰り返す一連の電気パルスとして送られます。クロックサイクルに応じて、各パルスはビットパターンを表す一連のビットに変換されます。

私たちはマザーボード上にいます。この一連のビットは、電気信号としてサウスブリッジを経て、8042マイクロコントローラーに送られます。このマイコンは、メイクコードをスキャンコードにデコードし、内部のレジスタに格納します。これが私たちのバッファです。内部レジスタはEEPROMチップなどで、いつでも電氣的にデータを上書きできるようになっています。

起動時には、BIOSのPOSTが各デバイス（I/Oコントローラーを除く）のポートアドレスを割り当てます。これは、デバイスをクエリーすることで行います。使用例では、BIOS POSTはこの内部レジスタをポートアドレス0x60に設定します。つまり、ポート0x60を参照するときは常に、この内部レジスタからの読み取りを要求していることになります。

ポートマッピングやIN/OUT命令についてはご存知の通りですので、そのレジスタから読み出してみましょう。

```
in al, 0x60
```

8042 マイコンの入力レジスタからのバイト取得

ご存知のように、8042マイクロコントローラーはキーボードコントローラーです。チップ内の様々なレジスタと通信することで、キーボードからの入力を読み取ったり、スキャンコードをマッピングしたり、その他様々なことができます。A20を有効にするように。

A20を有効にするために、なぜキーボードコントローラーに通信をしなければならないのか、疑問に思われるかもしれません。これについては次に説明します。

ゲートA20-理論

ついにA20を取り上げます。これまでのチュートリアルのはほとんどは、A20とは直接関係のない他のトピックを扱っていました。しかし、私は、A20に入る前に、まずハードウェアの直接プログラミングの基本から始めたかったのです。A20を有効にするには、マイクロコントローラーのプログラミングと同様に、ハードウェアの直接プログラミングが必要だからです。

A20ラインを有効にするには、キーボードマイクロコントローラーのプログラミングが必要になる場合があります。このため、キーボードコントローラーのプログラミングについては少し触れますが、キーボードのプログラミングについてはまだ触れません。

歴史を振り返る

IBM社が設計した「**IBM PC AT**」は、新しいマイクロプロセッサ「**インテル80286**」を採用していたが、このマイクロプロセッサは、従来のx86マイクロプロセッサと実使用時に完全な互換性がない。問題は？古いx86プロセッサには、アドレスラインA20～A31がない。そのサイズのアドレスバスがまだなかったのだ。最初の1MBを超えてプログラムを実行すると、回り込んで見えてしまう。当時の80286のアドレス空間は32本のアドレスラインを必要としていました。しかし、32本すべてにアクセスできるようにすると、再び回り込みの問題が発生してしまいます。

この問題を解決するために、インテルはプロセッサとシステムバスとの間の20番目のアドレスラインにロジックゲートを設置した。このロジックゲートは、有効にも無効にもできることから「**ゲートA20**」と名付けられた。古いプログラムではラップワウンドに依存するプログラムを無効にし、新しいプログラムでは有効にすることができます。

起動時、BIOSはメモリのカウントとテストの際にA20を有効にし、OSに制御権を与える前に再び無効にします。

A20を有効にするには、さまざまな方法があります。A20ゲートを有効にすることで、アドレスバスの32ラインすべてにアクセスできるようになり、32ビットのアドレス、つまり最大0xFFFFFFFF（4GBのメモリ）を参照できるようになります。

ゲートA20は、電子式のORゲートで、元々は、8042マイコン（キーボードコントローラ）のP21電気ラインに接続されていました。このゲートは、出力ポートのデータのビット1として扱われる出力ラインです。このデータを受信するコマンドを送信したり、データを変更したりすることができます。このビットを設定し、出力ラインのデータを書き込むことで、マイコンがORゲートを設定し、A20ラインを有効にすることができます。これは、私たち自身が直接、または間接的に行うことができます。詳細は次のセクションで説明します。

起動時には、BIOSがA20ラインを有効にしてメモリをテストします。メモリテストの後、BIOSは古いプロセッサとの互換性を保つためにA20ラインを無効にします。このため、当社のOSでは、デフォルトでA20ラインが無効になっています。

ゲートA20を再び有効にするには、マザーボードの設定によっていくつかの異なる方法があります。そのため、ここではA20を有効にするためのより一般的な方法をいくつか紹介します。

次はこれを見てみましょう。)

ゲートA20 - イネーブル

A20を有効にするには、さまざまな方法があることを覚えておってください。単にシステムを動作させたい場合は、自分に合った方法を使用すればよいでしょう。移植性が要求される場合は、複数の方法を併用する必要があるかもしれません。

方法1：システムコントロールポートA

これは非常に高速な方法ですが、A20アドレスラインを有効にするためのポータブルな方法ではありません。

MCAやEISAを含むSoneシステムでは、システムコントロールポートI/O 0x92からA20を制御することができます。0x92番ポートの詳細や機能は、メーカーによって大きく異なります。しかし、一般的に使用されるビットがいくつかあります。

- **ビット0** - 1にすると高速リセットがかかる（リアルモードに戻すのに使用する）
- **ビット1** - 0：A20を無効にする、1：A20を有効にする
- **ビット2** - メーカー定義
- **ビット3** - パワーオンパスワードバイト（CMOSバイト0x38-0x3fまたは0x36-0x3f）。0：アクセス可能、1：アクセス不可能
- **ビット4-5** - メーカー定義
- **Bits 6-7** - 00: HDD Activity LED off; any other value is "on"

この方法でA20を有効にした例を示します。

```

ム      al, 2; ビット2 を設定
一      0x92を有効にする)
ブ
ア
ウ
ト

```

このポートでできることは他にもたくさんあります。

```

ム      al, 1; ビット1 を設定 (高
一      速リセット)
ブ
ア
ウ
ト

```

この方法は、Bochsでも使えそうです。

警告!

これは簡単な方法の1つですが、この方法が他のいくつかのハードウェアデバイスと衝突することがあります。この方法では、通常、システムが停止してしまいます。この方法を使用したい（そしてそれがうまくいっている）場合は、私はこの方法を使用しますが、このことを覚えておいてください。

他のポート...

システムによっては、他のI/Oポートを使ってA20を有効にしたり無効にしたりすることができることをお伝えしておきます。

最も一般的なものは、I/Oポート0xEEです。これらのシステムでI/Oポート0xEE（「FAST A20 GATE」）が有効になっている場合、このポートから読み取るとA20が有効になり、書き込むとA20が無効になります。同様の効果は、システムをリセットするためのポート0xEF（「FAST CPU RESET」）にも生じます。

他のシステムでは、異なるポートを使用している場合があります（例：AT&T 6300+では、A20を有効にするにはI/Oポート0x3f20に0x90を書き込み、A20が無効にするには0を書き込む必要があります）。また、I/Oポート0x65のビット2やI/Oポート0x1f8のビット0を使ってA20を有効にするシステムが存在するという噂もあります。

とA20を無効にすることができます（0：無効、1：有効）。

このように、A20を使用する際には多くの問題があります。確実な方法は、マザーボードメーカーに問い合わせるしかありません。

方法2：Bios

多くのBiosは、A20を有効または無効にするために、割り込みを利用できるようになっています。

ボウズサポート

Bochsの一部のバージョンではこれらの方法を認識しているようですが、Bochsの一部のバージョンではサポートされていないかもしれません。

INT 0x15 Function 2400 - Disable A20

A20のゲートを無効にする機能です。使い方はとても簡単です。

```
mov ax, 0x2400
int 0x15
```

戻ります。

CF = 成功したらクリア

ア AH = 0

CF=エラー時に設定

AH = ステータス (01=キーボードコントローラがセキュアモードになっている、0x86=機能がサポートされていない)

INT 0x15 機能 2401 - イネーブル A20

この機能は、A20ゲートを有効にします。

```
mov ax, 0x2401
int 0x15
```

CF = 成功したらクリア

ア AH = 0

CF=エラー時に設定

AH = ステータス (01=キーボードコントローラがセキュアモードになっている、0x86=機能がサポートされていない)

INT 0x15 機能 2402 - A20 ステータス

この関数は、A20ゲートの現在のステータスを返します。

```
mov ax, 0x2402
int 0x15
```

戻ります。

CF = 成功したらクリア

AH = ステータス (01: キーボードコントローラがセキュアモードになっている; 0x86: 機能がサポートされていない) AL = 現在の状態 (00: 無効, 01: 有効)

CX = 0xffffに設定されるのは、キーボードコントローラが0xc000の読み出し試行でレディになっていない場合 CF = エラー時に設定される

INT 0x15 機能 2403 - 問い合わせ A20 サポート

この機能は、A20のサポートをシステムに問い合わせるために使用します。

```
mov ax, 0x2403
int 0x15
```

CF = 成功したらクリア

AH = ステータス (01: キーボードコントローラがセキュアモードになっている; 0x86: 機能がサポートされていない) BX = ステータス。

BXはビットパターンを含みます。

- **ビット0** - キーボードコントローラでサポートされている場合は**1** **ビット1** - I/Oポート0x92の**ビット1**
- でサポートされている場合は**1** **ビット2-14** - 予約
- **ビット15** - 追加データがある場合は**1**。

方法3：キーボードコントローラー

これは、おそらくA20を有効にする最も一般的な方法です。非常に簡単ですが、キーボードのマイクロコントローラーのプログラミングに関する知識が必要です。この方法が最もポータブルであると思われるため、私はこの方法を使用することにします。この方法にはキーボードマイコンのプログラミングに関する知識が必要なので、まずその点を少し見てみましょう。

これが、ハードウェア・プログラミングを最初に取り上げようとした理由でもあります。これは、直接ハードウェア・プログラミングに触れる最初の機会であり、それがどのようなものなのかを知るためのものです。心配しないでください、そんなに悪いものではありません。)時には複雑になることもあります.....。)

8043キーボードコントローラ - ポートマッピング

このコントローラと通信するためには、コントローラが使用しているI/Oポートを知っておく必要があります。このコントローラのポートマッピングは以下の通りです。

ポートマッピング		
ポート	リード/ライト	ディスクリプション
0x60	リード	リード入力バッファ
0x60	書く	ライト出力バッファ
0x64	リード	ステータスレジスタの読み出し
0x64	書く	コントローラへのコマンド送信

I/Oポート0x64にコマンドバイトを書き込むことで、このコントローラにコマンドを送ります。コマンドがパラメータを受け入れる場合、このパラメータはポート0x60に送信されます。同様に、コマンドによって返された結果は、ポート0x60から読み取ることができます。

ここで注意しなければならないのは、キーボードコントローラ自体がかなり遅いということです。私たちのコードはキーボードコントローラよりも高速に実行されるので、先に進む前にコントローラの準備が整うのを待つ方法を提供しなければなりません。

これは通常、コントローラの状態を問い合わせることで行われます。わかりにくいかもしれませんが、すぐに理解できると思います。

8043キーボードコントローラステータスレジスタ

さて、コントローラの状態を知るにはどうすればいいでしょうか？上の表を見ると、I/Oポート0x64から読み取る必要があることがわかります。このレジスタから読み出される値は、特定のフォーマットに従った8ビットの値です。ここでは...

- **ビット0：出力バッファステータス**
 - 0: 出力バッファが空、まだ読まないでください
 - さい 1: 出力バッファがいっぱい、読んでください :)
- **ビット1：入力バッファステータス**
 - 0: 入力バッファが空、書き込み可能
 - 1: 入力バッファが満杯、まだ書き込まないでください
- **ビット2：システムフラグ**
 - 0：パワーオンリセット後に設定
 - 1：キーボードコントローラのセルフテスト（Basic Assurance Test, BAT）が正常に終了したときに設定されます。
- **ビット3：コマンドデータ**
 - 0: 入力バッファへの最後の書き込みはデータ（ポート0x60経由）
 - 1: 入力バッファへの最後の書き込みがコマンド（ポート0x64経由）だった場合
- **Bit 4: キーボードロック**
 - 0: ロックされている
 - 1: ロックされていない
- **ビット5：Auxiliary Output buffer**
 - full PS/2 Systems:
 - 0：ポート0x60からの読み出しが有効かどうかを判断 有効な場合、0＝キーボードデータ 1：マウスデータ、ポート0x60からの読み出しが可能な場合のみ
 - ATシステム。
 - 0: OKフラグ
 - 1：キーボードコントローラからキーボードへの送信がタイムアウトした。これは、キーボードが存在しないことを示している可能性があります。
- **ビット6：タイムアウト**
 - 0: OKフラグ
 - 1: タイムアウト PS/2
 - 一般的なタイムアウト
 - AT:
 - キーボードからキーボードコントローラへの送信のタイムアウト。パリティエラーの可能性あり（この場合、ビット6と7の両方がセットされます
- **ビット7：パリティエラー**
 - 0：OKフラグ、エラーなし
 - 1：最終バイトのパリティエラー

2021/11/15 13:06

オペレーティングシステム開発シリー

ご覧のように、ここでは多くのことが行われています。重要な部分(上の太字の部分で、コントローラの出力または入力バッファが一杯になっているかどうかを示しています。

以下にその例を示します。コントローラーにコマンドを送るとします。このコマンドはコントローラの入力バッファに入ります。このバッファがいっぱいになっている間は、コマンドが実行されていることになります。コードは以下のようになります。

```
wait_input
です。    で    al,0x64    リード・ステータス・レジスタ
          テスト  al,2      テストビット2(入力バッファの状態)
          jnz    wait_input  0でなければ(空でなければ)ジャンプし、待機を続ける
```

これは、入力バッファと出力バッファの両方で必要となります。

コントローラを待つことができるようになったので、次は実際にコントローラに何をしてほしいかを伝えなければなりません。これは、コマンドバイトで行います。では、見てみましょう。

8043キーボードコントローラコマンドレジスタ

I/Oポート表を見ると、コントローラにコマンドを送るためにはI/Oポート0x64に書き込む必要があることがわかる。

キーボードコントローラには多くのコマンドがあります。この記事はキーボードプログラミングのチュートリアルではありませんので、ここではすべてのコマンドを紹介しません。しかし、より重要なものはリストアップします。

キーボードコントローラのコマンド	
キーボードコマンド	ディスクリプション
0x20	キーボードコントローラコマンドバイトの読み込み
0x60	Write Keyboard Controller Command Byte
0xAA	セルフテスト
0xAB	インターフェーステスト
0xAD	キーボードの無効化
0xAE	キーボードの有効化
0xC0	リード入力ポート
0xD0	リード出力ポート
0xD1	書き込み出力ポート
0xDD	A20アドレスラインの有効化
0xDF	A20アドレスラインの無効化
0xE0	リードテスト入力
0xFE	システムリセット
マウスコマンド	ディスクリプション
0xA7	マウスポートの無効化
0xA8	マウスポートの有効化
0xA9	テストマウスポート
0xD4	マウスへの書き込み

繰り返しになりますが、この他にも多くのコマンドがあります。後日、すべてをご紹介しますので、ご安心ください)

方法3.1:キーボードコントローラでA20を使用する

上の表のコマンドバイト0xDDと0xDFに注目してください。これはキーボードコントローラを使ってA20を有効にする一つの方法です。

すべての方法3.1:キーボードコントローラがこの機能に有効にするわけではありません。もし機能するのであれば、私はそのシンプルさにこだわっています。) 0xdd; command 0xdd: enable a20
out 0x64, al; コントローラへのコマンド送信

方法3.2: 出力ポートからA20を起動する

A20を有効にするもう一つの方法は、キーボードコントローラの出力ポートを利用することです。そのためには、コマンドD0とD1を使って、出力ポートの読み書きを行う必要があります。

この方法は、他の方法に比べて少し複雑ですが、それほど悪いものではありません。基本的には、キーボードを無効にして、コントローラから出力ポートを読み取ります。8042には3つのポートがあります。1つは入力、もう1つは出力です。そうですか...。3つ目はテスト用です。これらの"ポート"は、マイクロコントローラのハードウェアピンに過ぎません。

ここでは、シンプルにするために（そして、これはキーボードプログラミングのチュートリアルではないので）、出力ポートだけを見てみましょう。

さて、出力ポートからの読み込みですが、単純に出力ポート読み込みコマンド（0xD0）をコントローラに送ります。(参考までに、キーボードコントローラのコマンド表をご覧ください。)

```
; 出力ポートをalに読み込む
mov al,0xD0
アウト0x64,アル
```

これで、出力ポートのデータを得ることができました。いいですね^ズでも、これではあまり役に立ちませんよね？実は、出力ポートのデータは、またしても特定のビットフォーマットに従っているのです。

それでは見てみましょう。

- **ビット0**：システムリセット
 - 0：コンピュータのリセット
 - 1：通常動作
- **ビット1**：**A20**
 - 0：Disabled
 - 1：イネーブル
- **ビット2-3**：未定義
- **ビット4**：入力バッファフル
- **ビット5**：出力バッファエンブティ
- **ビット6**：キーボードク
 - ロック 0：High-Z
 - 1：プルクロックロー
- **ビット6**：キーボードデ
 - ータ0：High-Z
 - 1：プルデータロー

これらのビットのほとんどは変更したくありません。ビット0を1にするとコンピュータがリセットされ、ビット1にするとゲートA20が有効になります。他のビットに触れないように、この値をORしてビットを設定する必要があります。ビットを設定した後は、その値を書き戻すだけです（コマンドバイト0xD1）。

出力ポートの読み書きに使用されるコマンドは、データにコントローラの入力バッファと出力バッファを使用します。

つまり、出力ポートを読み取ると、読み取ったデータはコントローラの入力バッファレジスタに入ります。I/Oポート表を見てみると、これはI/Oポート0x60から読み出したデータを取得することを意味しています。

例を見てみましょう。**wait_input**は入力バッファが空になるのを待ち、**wait_output**は出力バッファが空になるのを待ちます。

```

; 出力ポートのリードコマンドを送信
mov al, 0xD0
アウト 0x64, アル
call wait_output

入力バッファを読み込み、スタックに格納します。これは、出力ポートから読み込まれたデータです
in al, 0x60
push
call wait_input

; 書き込み出力ポートコマンドの送信
mov al, 0xD1
アウト 0x64, アル
call wait_input
pop
出力ポートのデータをスタックからポップしてレジスタ1 (A20) をイネーブルにします。
アウト 0x60, al // 出力ポートにデータを書き込みます。これは、出力バッファを介して行われます

```

それが全てです。:)この方法は他の方法よりも少し複雑ですが、最もポータブルな方法でもあります。

注意点について

エミュレーションであるため、これらのほとんどはBochsではなく、実際のハードウェアに適用されます。

コントローラが誤ったコマンドを実行した

コントローラが間違ったコマンドを実行すると、通常は望んでいないことをしてしまいます。例えば、データの書き込みではなく、ポートからデータを読み出すなど、データが乱れる可能性があります。例えば、**in al, 0x60**ではなく、**in al, 0x61**を使用すると、ステータスレジスタ（ポート0x60）ではなく、キーボードマイコンの異なるレジスタから読み込まれます。

不明なコントローラコマンド

ほとんどのコントローラは、知らないコマンドを無視して破棄します（コマンドレジスタがあればクリアします）。しかし、コ

ントローラによっては誤動作することがあります。詳しくは、「誤動作」の項をご覧ください。

コントローラの誤動作

滅多にないことですが、可能性はあります。顕著な例としては、PentiumプロセッサのFDIVバグとfoofバグがあります。FDIVバグは、CPU内部の設計上の欠陥で、プロセッサ内部のFPUが誤った結果を出すというものです。

foofの問題はもっと深刻です。プロセッサにコマンドバイト0xf0 0xf0f 0xc7 0xc8が与えられたとき、これは**HCF (Halt and Catch Fire)**命令の一例です。**(An Undocumented Instruction)**と呼ばれます。これらの命令の多くは、プロセッサ自体をロックしてしまい、ユーザーが

ハードリブートを余儀なくされる可能性があります。また、これら~~安~~命令を使用することで、異常な副作用が発生するものもあります。

このような問題が起こる可能性があることを、よく考えてみる必要があります。それは、コントローラも例外ではありません。(コントローラも例外ではありません(命令バイトを個々のポートに送ることを覚えていますか?例えば、ポート0x64は、キーボードコントローラのコマンドレジスタです。))

これらの不具合のほとんどは、デバイスの「設計上の欠陥」と考えて差し支えありません。

物理的なハードウェアの損傷

まれなケースではありますが、ソフトウェアを使ってハードウェアにダメージを与えることも可能です。例えば、フロッピーディスクドライブです。フロッピードライブの**モーターは、FDC (Floppy Drive Controller)** で直接制御する必要があります。モーターを停止させるコマンドを送り忘れると、フロッピードライブが摩耗して壊れてしまいます。気をつけましょう。

トリプルフォールト

マイクロコントローラは、コントロールバスを介して問題があることをプライマリプロセッサに通知することができ、その場合、プロセッサは例外を通知し、当然ながらコンピューターを再起動します。

Bochsのコントローラの問題

コントローラに問題がある場合、BochsはTriple faultを発生させ、その情報(The problem)をログに記録します。例えば、キーボードコントローラに未知のコマンド(0など)を送ろうとしたとします。

movable アウト	al, 0x00 0x64, al	適当なコマンド コントローラへのコマンド送信を試みる
----------------	----------------------	-------------------------------

Bochsはトリプルフォールトを誘発し、その情報を記録する。

[KBD] キーボードポート64への サポートされていないIO書き込み、値=0

"KBD "は、キーボード・コントローラ・デバイスによってログが書き込まれたことを表します。

デモ

A20のコードはすべて**A20.inc**にあります。私は、A20を有効にするための異なる方法を使用するいくつかの異なるルーチンを書きました。ですから、ある方法が失敗したら、別の方法を使ってみてください。

複雑になってきたので、このデモをダウンロードできるようにしました。現在の**Stage2.asm**もそれほど大きくは変わっていません。

デモでは新しいものを表示しないため、表示する新しい画像もありません。最新のデモ(*.ZIP: 8KB)

は[こちらから](#)ダウンロードできます。

結論

ワオ。ただ、すごい。このチュートリアルは、私が最初に予想したよりも大きいです。

ここでは、多くの新しいコンセプトを検討しました。また、ハードウェア・プログラミングも体験しました。覚えておいてください。**プロテクトモードでハードウェアと通信するには、この方法しかありません**。割り込みとはおさらばです。さようなら、BIOS、さようなら。全てに別れを告げて、私たちは完全に自分の力で生きていくのです。

今なら、Windowsをもう少し評価してもいいかもしれませんね。)結局のところ、彼らは皆、私たちのレベルから始めなければならなかったのです。

まだすべてを理解していなくても気にしないでください--複雑なのはわかっています。カーネルが完成したら、キーボードのマイクロコントローラをプログラミングして、そのためのドライバを書くことに特化したチュートリアル全体を用意しています。いいですか?

次のチュートリアルはもっと簡単になります。**Protected Mode**は一旦保留にして、**Real Mode**のコードに戻ります。カーネルをロードするための**FAT12**ロードコードを追加します。**A20**が有効になったので、**1MB**でロードできるようになりました!

また、BIOSの情報やその他思いついたことをお伝えします :)それではまた。次の機会まで。

マイク

BrokenThorn Entertainment 社。現在、DoEとNeptune Operating Systemを開発中です。質問

やコメントはありますか?お気軽に[お問い合わせ](#)ください。

あなたも記事の改善に貢献したいと思いませんか?もしそうなら、ぜひ[私に教えてください](#)。





オペレーティングシステム開発シリーズ

オペレーティングシステム開発 - カーネルの準備 part 1

by Mike, 2008

このシリーズは、オペレーティングシステムの開発を一から実演し、教えることを目的としています。

はじめに

歓迎します。:)

ここまで、いろいろと説明してきましたが、いかがでしたでしょうか？OS開発の複雑さがおわかりいただけたと思います。しかし、それはさらに難しくなる一方です。

今回は2部構成のチュートリアル第1回目です。第1部では、新しいコードのすべてを詳しく説明します。アセンブリによる基本的な32ビットグラフィックスのプログラミングについて説明します。これには以下が含まれます。基本的なVGAプログラミングのコンセプト、ビデオディスプレイへのアクセス、文字列の印刷、スクリーンのクリア、ハードウェアカーソルの更新などです。多少の計算はありますが、それほど多くはありません:)

デモ自体は完成しています。このチュートリアル第2部では、完成したステージ2のソースコードの概要と、その新しい小さなFAT12ドライバ、フロッピードライバを含めて紹介します。これらは、私たちが追加する定義上の「本物」のドライバではありません。しかし、これらはドライバの機能とその有用性を説明するのに役立ちます。すべてのコードは、ブートローダからのFAT12ロード・コードを大幅に修正したものですので、FAT12についての詳細な説明はしません。

第2部では、ステージ2の最後のチュートリアルとして、1MBの基本的な（純粋な）カーネルイメージのロードと実行について説明します。

この2部構成のチュートリアルは、ステージ2の最後のチュートリアルとなりますカーネルを起動する際には、さまざまな実行形式のファイルについて説明する必要があります。ステージ2がオブジェクトファイルを正しく実行できるようにしなければなりません。このため、カーネルを起動する際には、現在のStage 2のブートローダーにローダーを追加して、カーネルが正しくロードされるようにします。これは後の話ですが:)

これらを踏まえた上で、このチュートリアルのパート1ではVGA

- プログラミングの基本概念
- ディスプレイへのアクセス
- 文字の印刷 文字列の印刷
- CRT マイコンの理論とハードウェアカーソルの更新 画面の消去
-

このチュートリアルは、「[悪名高いチュートリアル7](#)」を参照しています。つまり、リアルモードアドレスマッピングとデフォルトI/Oポートアドレスです。ビデオアドレス空間やVGAポートのアクセスについて話すときに、このチュートリアルを参照すると役に立つかもしれません。

いいですか？

ディスプレイ

VGA - 理論

VGA (Video Graphics Array) は、1987年にIBMが発売したアノログコンピュータディスプレイ規格。アレイ」と呼ばれるのは、もともとMDA、CGA、EGAが使用していたISA (Industry Standard Architecture) ボードの数十個のロジックチップに代わる1つのチップとして開発されたからである。ISAボード1枚に収まっていたため、マザーボードへの接続が非常に容易でした。

VGAは、ビデオバッファ、ビデオDAC、CRTコントローラ、シーケンサユニット、グラフィックコントローラ、属性コントローラで構成されています。ここでは、ビデオドライバの話始めるまで、すべてを詳しく説明しないことに注意してください。これは、スペースを確保するためと、VGAのプログラミングが非常に複雑になるため、より簡単にするためです。

ビデオバッファ

ビデオバッファは、ビデオメモリとしてマッピングされたメモリのセグメントです。メモリのどの領域がビデオメモリにマッピングされるかは変更できる。起動時には、BIOSが0xA0000にマッピングしているので、ビデオメモリは0xA0000にマッピングされていることになります。(チュートリアル7のリアルモードアドレスマッピングを覚えていませんか?) これは重要です!

ビデオDAC

Video Digital to Analog Converter (DAC) には、ビデオデータをディスプレイに送るアナログビデオ信号に変換するためのカラーパレットが含まれています。この信号は、赤、緑、青の色の濃さをアナログで表したものです。詳しくは後で説明しますので、まだ理解できなくてもご安心ください。

CRTコントローラ

このコントローラは、水平および垂直同期信号のタイミング、ビデオバッファのアドレスマッピング、カーソルおよびアンダーラインのタイミングを生成します。カーソルを更新する際にCRTコントローラを経由する必要があるため、このチュートリアルの後半で詳細を説明します。

シーケンサー

シーケンサは、ビデオメモリの基本的なメモリタイミングと、再生バッファのフェッチを制御するためのキャラクタクロックを生成します。これにより、システムはアクティブなディスプレイのインターバル中にメモリにアクセスすることができます。もう一度言いますが、ここではまだ詳しく説明しません。後日、ビデオドライバの項で詳しく説明しますので、ご安心ください:)

グラフィックスコントローラ

ビデオメモリとアトリビュートコントローラ、ビデオメモリとCPUの間のインターフェースである。表示がアクティブな時間帯には、ビデオバッファ（ビデオメモリ）からメモリデータが送られ、アトリビュートコントローラに送られる。グラフィックスモードでは、このデータをパラレルからシリアルビットプレーンデータに変換してから送信する。テキストモードでは、パラレルデータだけが送信されます。

まだ理解できなくても大丈夫です。ここでは、あまり詳しく説明するつもりは**ありません**。後日、ビデオドライバの開発について説明するときに、すべてを詳しく説明します。とりあえず、覚えておいてください。**グラフィックスコントローラは、ビデオメモリからのパラレルデータをもとにディスプレイを更新します**。これは、ディスプレイの使用時間に応じて自動的に行われます。つまり、**ビデオメモリ（デフォルトは0xA0000にマッピングされています）に書き込むことで、現在のモードに応じて、実質的にビデオディスプレイに書き込むことになります**。これは、文字を印刷するときに重要です。

グラフィックス・コントローラが使用するアドレス範囲を変更することが可能であることを覚えておいてください。初期化の際、BIOSはビデオメモリを0xA0000にマッピングするためにこれを行います。

ビデオモード

ビデオモード」とは、表示の仕様である。つまり、**ビデオメモリ**がどのように参照され、そのデータがビデオアダプターでどのように表示されるかを記述したものです。

VGAは2種類のモードに対応しています。**APAグラフィックス**と**テキスト**です。

APAグラフィックス

APA (All Points Addressable) とは、ビデオモニターやドットマトリックスなど、ピクセルアレイで構成されたデバイスにおいて、すべてのセルを個別に参照できる表示モードのことである。ビデオディスプレイの場合は、すべてのセルが「ピクセル」を表し、すべてのピクセルを直接操作することができます。そのため、ほとんどのグラフィックモードがこの方式を採用しています。**このピクセルバッファを変更することで、画面上の個々のピクセルを効果的に変更することができます**。

ピクセル

ピクセル」とは、ディスプレイ上で表現できる最小単位のこと。ディスプレイ上では、色の最小単位を表しています。つまり、基本的には1つのドットである。各ピクセルのサイズは、現在の解像度とビデオモードに大きく依存します。

テキストモード

テキストモードとは、APAのように、画面上のコンテンツを内部的にピクセルではなく文字で表現する表示モードです。

テキストモードを採用しているビデオコントローラでは、**2つのバッファ**を使用します。**1つは、表示される各文字のピクセルを表すキャラクターマップ**、もう**1つは、各セルにどのような文字があるかを表すバッファ**です。文字マップバッファを変更することで、文字そのものを変更ことができ、新しい文字セットを作成することができます。また、各セルにどのような文字が入っているかを表す**スクリーンバッファ**を変更することで、**画面に表示される文字を変更することができます**。テキストモードの中には、文字の色や、点滅、下線、反転、明るくするなどの属性を設定できるものもあります。

MDA, CGA, EGA

VGAはMDA、CGA、EGAをベースにしていることを忘れてはならない。**VGAはこれらのアダプタが行うモードの多くをサポートしています**。これらのモードを理解することで、VGAの理解が深まります。

MDA-理論

私が生まれる前（真面目な話）の1981年、IBMはPC用の標準的なビデオディスプレイカードを開発した。それが「**モノクロディスプレイアダプター（MDA）**」と「**モノクロディスプレイ・プリンタアダプター（MDPA）**」である。

MDAには、グラフィックモードは一切ない。**唯一のテキストモード（モード7）では、80列×25行の高解像度のテキスト文字を表示することができた**。

このディスプレイアダプターは、古いPCで使われていた一般的な規格でした。

CGA - 理論

1981年には、IBMが**CGA (Color Graphics Adapter)**を開発し、PCの最初のカラーディスプレイ規格となりました。CGAは、**1ピ**

クセルが4バイトに制限されていたため、**16色のカラーパレット**しかサポートしていなかった。

CGAは、以下の2つのテキストモードと2つのグラフィックモードをサポートしていました。

- 40x25文字（16色）テキストモード 18x25文字
- （16色）テキストモード 320x200ピクセル（4
- 色）グラフィックモード
- 640x200ピクセル（モノクロ）グラフィックモード

ディスプレイアダプタを使って、「文書化されていない」新しいビデオモードを作成したり、発見したりすることができます。これについては後で詳しく説明します。

EGA - 理論

1984年にIBMが発表した**EGA (Enhanced Graphics Adapter)**は、最大640×350ピクセルの解像度で**16色のディスプレイ**を実現した。

VGAアダプタは、80x86マイクロプロセッサ・ファミリーと同様に下位互換性があることを覚えておいてください。このため、後方互換性を確保するために、**BIOSは80列×25行をサポートするモード7（MDAに由来する）で起動します**。これは、私たちにとって重要なことです。

VGAメモリのアドレス指定

VGAコントローラが使用するビデオメモリは、PCのメモリの0xA0000から0xBFFFFまでにマッピングされます。**チュートリアル7のリアルモードメモリーマップ**を思い出してください。

通常、ビデオメモリーは以下のようにマッピングされています。

- **0xA0000 - 0xBFFFF** グラフィックスモードで使用するビデオメモリ
 - **0xB0000 - 0xB7777** モノクロテキストモード
 - **0xB8000 - 0xBFFFF** カラーテキストモード、CGA互換グラフィックモード

メモリマッピングで使用するアドレスが異なるため、ECG、CGA、VGAの各ディスプレイアダプターを同じマシンにインストールすることが可能です。

CRTマイコンを介して、**ビデオアダプターカードが使用するメモリマッピングを変更することができます**。通常、これはビデオドライバで行います。これについては後で詳しく説明します。

また、ビデオコントローラがこのメモリをどのように使用するかを変更することもできます。そうすることで、「新しい」というか、「文書化されていない」モードを作り出すことができます。一般的なモードとしては、悪名高い「モードX」があります。

ディスプレイバッファとテキストバッファを変更すると、画面に表示される内容が実質的に変わることを覚えていますか？これは、ビデオコントローラが現在のリフレッシュレートに基づいてディスプレイを更新しているためです。ビデオコントローラは、VGAポートを通じてモニター内のCRTコントローラにコマンドを送ります。これにより、CRTの垂直・水平方向のリトレースが行われ、モニターの表示が更新されます。そして、上記のPCのメモリアドレスには、テキスト・ディスプレイアダプターがマッピングされているので

メモリのこの領域に書き込むと、画面に表示される内容が変わります。

例えば、モード7であることを思い出してください。モード7はカラーテキストモードであるため、0xB8000から始まるメモリを使用します。これは、ビデオコントローラが表示内容を決定するために使用するテキストバッファなので、0xB8000に書き込むことで、画面にテキストが表示されます。

```
%define VIDMEM 0xB8000          ビデオメモリ

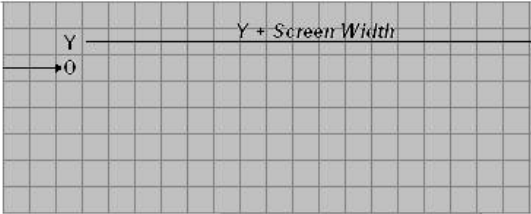
movab    EDI, VIDMEM            ビデオメモリへのポインタの取得
le       [edi], 'A'              ; 文字'A'を表示
movab    [edi+1], 0x7            文字属性
le
```

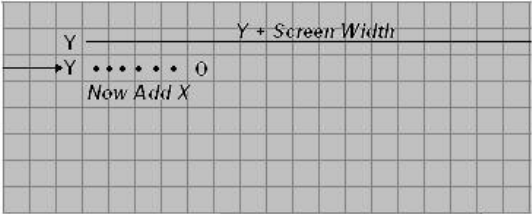
上記は、ディスプレイの左上に、白地に黒の背景（The attribute）で、文字「A」を表示します。かつこよすぎて学校では使えません :)

文字の印刷

では、画面上の任意のx/y位置に文字を印刷するにはどうすればよいのでしょうか。

メモリの特別な特性は、それがいかに直線的であるかということです。表示されている行の終わりに到達すると、次のバイトはそのすぐ下の行にあります。線形アドレスのため、画面に表示するためには、x/yの位置を線形アドレスに変換できなければなりません。そして、それを行うための特別なフォーミュラーは、「**x + y * 画面幅**」です。





Notice by multiplying screen width by a value of Y, we effectively move down one row on the screen within video memory.

Because of this, multiplying it by a value of Y means that, by incrementing the value of Y, we effectively go down Y rows.

Now, we add the value of "X" to this location, giving us a formula: **Location = x + (y * screen width)**

Remember that memory is linear!

以下にその例を示します。例えば、「A」という文字をx/y(5,5)の位置に印刷したいとします。ビデオメモリは0xb8000から始まり、直線的であることを知っているのので、このx/yの位置を絶対的なアドレスに変換する式を使うことができます。

つまり、0xB8195に文字「A」を書き込むことで、実質的にx/y位置(5,5)に書き込むことができます。すごいでしょ？

これを踏まえて、まずは画面上の現在地を保存する方法を提供します。これは、BIOSのように動作させるためのもので、他のプログラムには必要ありません。

アドレス = 405

```
CurX db 0
CurY db 0

%define VIDMEM 0xB8000          ビデオメモリ
%define 0xB8000 + 405 = 0xB8195 画面の幅と高さ
%define LINES 25
%define CHAR_ATTRIB 14          文字属性(黒地に白文字)
```

モード7であることを忘れないでください。このモードでは、1列の文字数が80コルム、行数が25行です。そしてもちろん、ビデオメモリは0xB8000から始まります。でも、待ってください。文字属性とは何でしょうか？

テキストモード7では、1文字あたり1バイトではなく、実際には2バイトを使用します。これを覚えておいてください。1バイト目は実際の文字を表し、2バイト目は.....ちよっと待って.....属性バイトです。このため、モード7で画面に文字を書き込む際には、1バイトではなく2バイトで書き込む必要があります。

アトリビュートバイトでは、色や、点滅などの特定の属性を指定することができます。値は、...

- 0 - ブラック
- 1 - ブルー
- 2 - グリーン
- 3 - シアン
- 4 - レッド
- 5 - マゼンタ
- 6 - ブラウン

- **7** - ライトグレイ
- **8** - ダークグ
- レー **9** - ライトブルー
- **10** - ライトグリー
- **11** - ライトシア
- **12** - ライトレッド
- **13** - ライトマゼンタ
- **14** - ライトブラウン
- **15** - ホワイト

属性バイトは、特定の属性を定義するバイトで、前景色と背景色を定義します。このバイトは以下のフォーマットに従います。

- **ビット0〜2**：フォアグラウンド・カラー
 - ビット0：赤
 - ビット1：グリーン
 - ビット2：ブルー
- **ビット3**：フォアグラウンド輝度 (Foreground Intensity)
- **ビット4〜6**：背景色
 - ビット4：赤
 - ビット5：グリーン
 - ビット6：ブルー
- **ビット7**：点滅または背景の強度

さて、すべての設定が完了したところで、キャラクターを印刷してみましょう。

設定方法

文字の印刷は、現在のx/y位置とビデオメモリへの書き込み時の両方で、位置を追跡する必要があるため、少し複雑です。また、改行文字などの特定の文字を追跡したり、行末を監視する必要もあります。さらに、ハードウェアカーソルをこの位置に更新する必要もあります。

Putch32は、ステージ2で文字を表示するpmodeルーチンです。心配しなくても、これらのルーチンをC言語を使ってカーネル用書き換えます。アセンブリでどのように行われているかを示すことで、アセンブリ言語の関係をC言語と比較することができます。

とにかく、スタートアップのコードです。

```
ビット 32

%define    VIDMEM    0xB8000          ビデオメモリ
%define    COLS      80              画面の幅と高さ
%define    LINES     25
%define    CHAR_ATTRIB 14            文字属性 (黒地に白文字)

_CurX db 0                          現在のX/Y位置
_CurY db 0

;*****;
;Putch32 ()
; - 画面に文字を表示する
;BL => 印刷する文字
;*****;

Putch32。

プーシ レジスタの保存
ヤ EDI、VIDMEM ビデオメモリへのポインタの取得
movabl e
```

さて、いくつかの基本的な定義があります。_CurXと_CurYには、文字を書き込むための現在のx/y位置が入ります。_CurXをインクリメントすることで、実質的に次の文字に進みます。また、EDIにはビデオメモリのベースアドレスが含まれています。これで、ビデオメモリ[EDI]に書き込むことで、現在のビデオメモリマップに合わせて画面に文字を表示することができます。

文字を表示する前に、どこに表示するかを調べなければなりません。そのためには、現在のx/y位置(_curXと_curY)に書き込みばいい。しかし、これはなかなか単純ではない。

ご存知のように、ビデオメモリはリニアなので、x/yの位置をリニアなメモリに変換する必要があります。**x + y * 画面幅**という式を思い出してください。これは簡単に計算できます。しかし、**すべての文字は2バイトの大きさであることを覚えておいてください**。_curX、_curY、COLS、LINESはバイトではなく、文字を基準にしていることを覚えておいてください。1文字が2バイトなので、80*2と比較しなければなりません。簡単でしょう？

そのため、少し複雑になっていますが、それほど難しいことはありません。

```
;-----;
; 現在の
; 位置
; を得る
xor     eax, eax          eaxのクリア
;-----;
;
; Remember: currentPos = x + y * COLS!xとyは_CurXと_CurYになります。
; 1文字が2バイトであることから、COLS=1行の文字数。
; これを2倍すると1行あたりのバイト数になります。これが画面の幅になります。
; つまり、スクリーンと* _CurYを掛け合わせて現在のラインを得る。
;-----;

movabl  ecx, COLS*2        モード7では、1文字が2バイトなので、1行あたりCOLS*2バイトとなります。
e
movabl  al, byte [_CurY].  ゲットイットポス
```

乗
マルチ ecx
プッシ eax
ユ

ズ
Y*COLSを掛け合わせる
EAXを保存して乗算を行う

これが式の最初の部分です。 **y * 画面の幅 (バイト)**、または **_curY * (COLS*1文字あたりのバイト数)** です。これをスタックに格納することで、式を完成させることができました。

```

;-----
; これでy *画面の幅がeaxになりました。あとは_CurXを追加するだけだ。しかし、_CurXは相対的なものであることを忘れてはい
けない。
は、バイト数ではなく、現在の文字数を表しています。1文字は2バイトなので
まず、_CurXに2をかけて、それを画面の幅*yに加える必要があります。
;-----
movab    al, byte  [_CurX].      _CurXに2をかけます。なぜなら、1つのcharが2バイトだからです。
le
movab    cl, 2
le
マルチ   cl
ポップ    ecx                    ポップY*COLSの結果
追加     eax, ecx

```

それじゃ、いいですか？現在のバイト位置を得るために、_CurXに2を掛けていることに注目してください。そして、**y * COLS**の結果をポップして**x**の位置に追加し、**x+y*COLS**の式を完成させています。

やったーさて、**EAX**には文字を表示するためのオフセットバイトが含まれているので、これを**EDI** (ビデオメモリのベースアドレス) に加えます。

```

;-----
; ここでeaxには文字を描画するためのオフセットアドレスが入っているので、これをベースアドレスに加えればよい。
ビデオメモリの容量 (エディに格納)
;-----
xor      ecx, ecx
add      edi, eax                    それをベースアドレスに加える

```

さて、**EDI**には書き込むべき正確なバイトが入っています。**BL**には書き込むべき文字が含まれています。もし、その文字が改行であれば、次の行に移動します。それ以外の場合は、単にその文字を表示します。

```

;-----;
新しい
    ラインに
    注目してください。
jmp      BL, 0x0A                ; は、改行文字ですか？
;-----;
; 次
; の行に進みます。
;-----;
; 1
; 文字
; を表示する
movab    dl, bl                  ; キャラクターの取得
;-----;
; 文字属性
movab    dh, CHAR_ATTRIB
le
movab    ワード[エディ], dx      ; ビデオディスプレイへの書き込
;-----;
; 次の
; ポジション
; を更新します
;
;-----;
inc      バイト  [_CurX]        ; 次の文字に移動
cmp      [_COPYX], COLS        ; 私たちは終着点にいるのでしょうか？
;-----;
; 次の行に進む。
jmp      .done                  ; nope, bail out

```

それじゃ！かなり簡単でしょう？そうですね、次の列に行くのも簡単です。

```

;-----;
次の 行に
    進みます。
;-----;
.Row:
mov      バイト  [_CurX],      コル0に戻る
inc      0 バイト  [_CurY]    次の行に進む
;-----;
; レジスターを      復元して戻る。
;-----;
; されて
; います
;
mov      ポパ
        レト                  レジスターを復元して戻る

```

ストリングスの操作

よし、これで文字を印刷できるぞ。イッパイです。一文字を見るのはとても楽しみです。ええ、そうは思いませんが :)

実際の情報を印刷するには、完全な文字列を印刷する方法が必要になります。すでに、現在の位置を追跡し (更新し)、文字を印刷するルーチンがあるので、文字列を印刷するために必要なのは、単純なループだけです。

```
Puts32
です。
;-----;
; レジスタの
; 保存
;-----;
```

プーシ		レジスタの保存
ヤブッシ	ebx	文字列アドレスのコピー
ユポップ	edi	

それでは、`Puts32()`関数をご紹介します。この関数は1つのパラメータを受け取ります。`EBX`は印刷するためのヌル終端文字列のアドレスを含んでいます。`Putch32()`関数では、印刷する文字を`BL`に保存する必要があるため、`EBX`のコピーを保存する必要があり、ここでそれを行います。

今はループしています。

```
.ループ
です。
;-----;
;
;   キャラクター
;   を取得する。
;-----;
movab    bl, byte [EDI].      次の文字を取得する
le
cmp      bl, 0                0 (Null terminator) なのか?
jz       .done                はい、救済します。
```

`EDI`を使用して文字列をデリファレンスし、表示する現在の文字を取得します。ヌル・ターミネーターのテストに注目してください。見つかった場合は処理を中止します。さて、その文字を表示するには…。これまでに見たこともないような複雑なコードです。

```

;-----;
;
;   その
;   文字
;   を印刷する。
;-----;
;   ユニコード
ラッチ32
; ノープリントアウト
```

…でもないか。)

あとは、次の文字に向かって、ループするだけです。

```

;-----;
;
;   次の 文字に
;   移動します。
;-----;
;
;   次に
;
;   イン      edi      次の文字に移動
;   ク        .ルー
;   ル        プ
;   ;-----;
;   ;   ハードウェア      カーソルの 更新
;   ;-----;
;   JMP
;
;   ; 表示後にカーソルを更新した方が効率的です。
;   ;   ダイレクトVGAでは遅いので、完全な文字列
;
;   mov      bh ,    バ イ ト      現在の位置を取得する
;   mov      [_CurY]  bl, バイ
;   call     ト      [_CurX]      カーソルの更新
;   MovCur
;
;   ボバ
;   レト      レジスタをリストアして、リターン
```

Voila! 文字列を32ビットのプロテクトモードで表示する方法ができました。難しいことではないでしょう？ちょっと待ってください。`MovCur`は何のためにあるの？次の機会に見てみましょう。

ハードウェアカーソルの更新

さて、これで文字や文字列をプリントアウトできるようになりました。しかし、カーソルが動かないことにお気づきでしょうか？このため、何をしてもカーソルは動かないのです。このカーソルは、`BIOS`がテキストを印刷する際に現在の位置を示すために使用する単純な下線です。

このカーソルはハードウェアで処理されます。実際には**CRTマイクロコントローラ**です。そのため、このカーソルを動かすためには、基本的な**VGA**プログラミングを知っておく必要があります。

CRTマイクロコントローラ

CRTユーザーへの警告

練習して新しいことに挑戦することは奨励しますが、OS環境ではハードウェアを直接操作することになるので、すべてを直接コントロールできることを忘れないでください。

CRTモニターの故障は激しく、爆発して鋭いガラスの破片が高速で飛び散ることがあります。周波数の設定を機器の許容範囲を超えて変更してしまうことがあります。これにより、デバイスやマイクロチップが誤動作する可能性が高まり、予測できない、または悲惨な結果を招くことがあります。

このため、読者の皆様がコードで実験をしたい場合は、実際のハードウェアに挑戦する前に、まずエミュレータで実験用のコードを最大限にテストすることをお勧めします。

ビデオドライバの話をするまでは、ビデオプログラミングに関するすべてを説明するつもりはありません。その時にすべてを詳しく見ていきますよ。とにかく…CRTコントローラの話をしてしまおう。

CRTコントローラは、ポート**0x3D5**にマッピングされた**1**つの**データレジスタ**を使用します。チュートリアル**7**のポートテーブルを覚えていますか？CRTコントローラは、データレジスタ内のデータのタイプを決定するために、特別なレジスタである**インデックスレジスタ**を使用します。

インデックスレジスターは、ポート0x3D5または0x3B5にマッ
ピングされます。データレジスターは、ポート0x3D4または
0x3B4にマッピングされます。

この他にも様々なレジスタ（Misc.Output Registerなど）がありますが、ここではこの2つに絞って説明します。

インデックスレジスタマッピング

デフォルトでは、インデックスレジスターのインデックスは以下のようにマッピングされています。

CRTマイクロコントローラ - インデックスレ ジスタ	
インデックスオ フセット	CRTコントローラレジスタ
0x0	水平方向の合計
0x1	Horizontal Display Enable End
0x2	水平ブランキング開始
0x3	エンドホリゾンタルブランキング
0x4	水平リトレースパルスの開始
0x5	エンドホリゾンタルリトレース
0x6	垂直方向の合計
0x7	オーバーフロー
0x8	プリセットロウスキャン
0x9	最大スキャンライン
0xA	カーソルスタート
0xB	カーソル終了
0xC	スタート・アドレス・ハイ
0xD	スタートアドレスLow
0xE	カーソル位置 高
0xF	カーソル位置 低
0x10	垂直リトレース開始
0x11	垂直リトレースエンド
0x12	バーティカルディスプレイイネーブルエ ンド
0x13	オフセット
0x14	下線部の位置
0x15	垂直ブランキング開始
0x16	エンドバーティカルブランキング
0x17	CRTモードコントロール
0x18	ライン比較

インデックスレジスタにインデックスオフセット値を書き込むことで、データレジスタがどのレジスタを指しているのか（つまり、何を参照しているのか）を示します。

上の表に書かれていることのほとんどは、今すぐに気にする必要はありません。しかし、0xEと0xFのインデックスに注目してみましょう。

- **0x0E:** カーソル位置ハイバイト
- **0x0F:** Cursor Location Low Byte

Yippe!これらのインデックスは、ハードウェアカーソルの現在のオフセット位置を参照しています。このオフセットは、x/yの位置（直線的な位置として、**x + y ***
画面の幅という公式を思い出してください！）を上位バイトと下位バイトに分割したものです。

ハードウェアカーソルの移動

まず、カーソルのインデックスが0x0Eと0x0Fであることを覚えておき、これをポート0x3D4のインデックス・レジスターに入れる必要があります。

```
mov     al, 0x0f
mov     dx, 0x03D4
out     dx, al
  
ブ
ア
ウ
ト
```

これにより、インデックス0x0F（カーソルの下位バイトアドレス）がインデックスレジスタに入ります。これで、データレジスタ（ポート0x3d5）に入れられた値は、
カーソル位置の下位バイトを示していることになります。

```
movab   アル、ブラ           alは下位バイトのアドレス
le      dx, 0x03D5
movab   dx, al
le      アウト               下位バイト
```

これで、カーソルの新しいローバイトの位置が設定されました。いいでしょう？上位バイトの設定も全く同じですが、インデックスを0x0Eに設定しなければなりません。こ
れは、やはり上位バイトのインデックスです。

これが完全なルーティンです。

ズ

```
;*****;  
;MoveCur ()  
; - ハードウェアカーソルの更新  
;parm/ bh = y pos  
;parm/ bl = x pos  
;*****;
```

```
xor     eax, eax
movab   ecx, COLS
le      al, bh           ; ギットイットボス
movab   ecx, Y*COLS      ; Y*COLSを掛け合わせる
le      ecx, ebx         ; 位置レジスタを保持する
movab   ebx, eax         ; 位置レジスタを保持する
le      dx, 0x03D4       ; CRTインデックスレジスタへの書き込み
movab   dx, al           ; 位置レジスタを保持する
le      dx, al           ; 位置レジスタを保持する

; ここで、_CurXと_CurYは、メモリ上ではなく、画面上の現在の位置に関連しています。
; また、文法を考慮するときのバイトアライメントを気にする必要がないので、デフォルトで、次のようになります。 location = _curX + _curY * COLS
movab   dx, 0x03D4       ; CRTインデックスレジスタへの書き込み
le      dx, al           ; 位置レジスタを保持する
movab   dx, al           ; 位置レジスタを保持する
le      dx, al           ; 位置レジスタを保持する

movab   al, bh           ; 現在の位置はEBXになります。BLには下位バイト、BHには上位バイト
le      eax, 0x03D5       ; VGAレジスタへのローバイトインデックスレジスタへの書き込み
movab   dx, 0x03D5       ; データレジスタへの書き込み
le      dx, al           ; 位置レジスタを保持する
movab   dx, al           ; 位置レジスタを保持する
le      dx, al           ; 位置レジスタを保持する

xor     eax, eax
movab   al, 0x0e         ; カーソル位置の上位バイトインデックス
le      eax, 0x03D4       ; VGAレジスタへのハイバイトインデックスレジスタへの書き込み
movab   dx, 0x03D4       ; データレジスタへの書き込み
le      dx, al           ; 位置レジスタを保持する
movab   dx, al           ; 位置レジスタを保持する
le      dx, al           ; 位置レジスタを保持する

movab   al, bh           ; 現在の位置は EBX になります。BLは下位バイト、BHは上位バイト
le      eax, 0x03D5       ; VGAレジスタへのローバイトインデックスレジスタへの書き込み
movab   dx, 0x03D5       ; データレジスタへの書き込み
le      dx, al           ; 位置レジスタを保持する
movab   dx, al           ; 位置レジスタを保持する
le      dx, al           ; 位置レジスタを保持する

movab   al, bh           ; 現在の位置は EBX になります。BLは下位バイト、BHは上位バイト
le      eax, 0x03D5       ; VGAレジスタへのローバイトインデックスレジスタへの書き込み
movab   dx, 0x03D5       ; データレジスタへの書き込み
le      dx, al           ; 位置レジスタを保持する
movab   dx, al           ; 位置レジスタを保持する
le      dx, al           ; 位置レジスタを保持する
```

簡単だったでしょう？
次の記事画面をスッキリさせたい

画面の消去

テキストを表示する方法はすでにあるので、ただループして、現在の位置を0にリセットするだけです。これは意外と簡単ですね。

簡単でしょう？*****;
ClrScr32 ()
これで、テキストを印刷し、画面を消去し、画面を更新し、画面を消去する方法ができました。欲を言えば、このステージ2のロードを拡張して、カーネルに制御を与える際の小さなメモリーや高度なオプションを含めることもできます。これについては後で説明します。

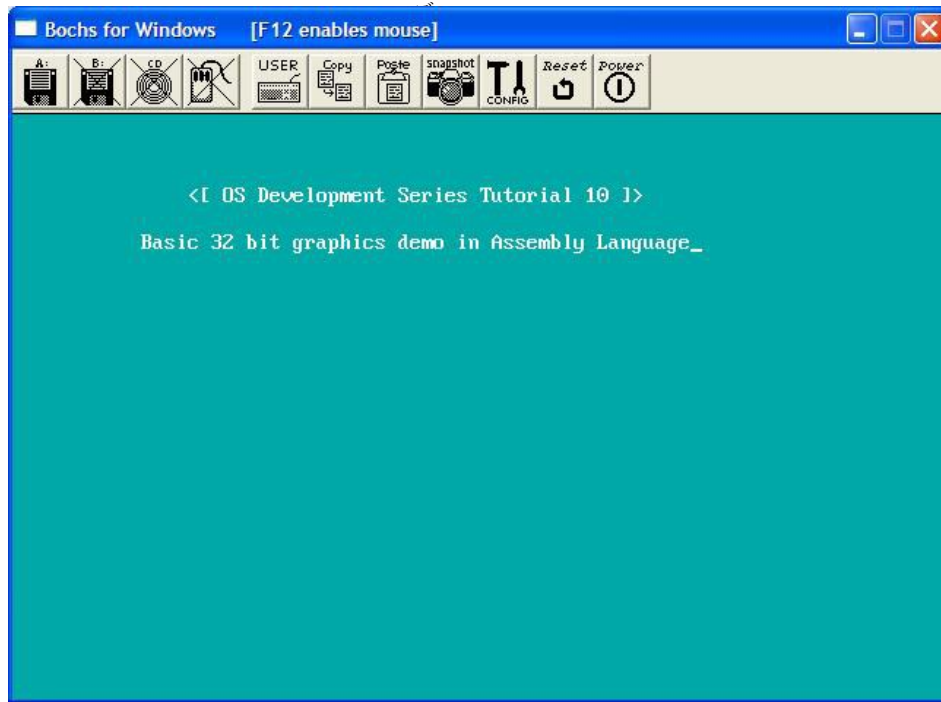
デモ

```
ClrScr32です。

PUSHA
CLD

movedi, VIDMEM
movcx, 2000
movah, CHAR_ATTRIB
ムーバ、' '
repstosw

movbyte [ CurX], 0
movbyte [ CurY], 0 popa
レット
```



このチュートリアルの内容を実演するために、小さなデモを作ることになりました。次のチュートリアルは、このコードをベースにしています。

このチュートリアルでは、このチュートリアルで説明した内容をすべて使用します。このチュートリアルでは、文字属性バイトに基づいて前景色と背景色を設定します。そして、`ClrScr32()`ルーチンのおかげで、効果的に画面を背景色にクリアしています。カッコいいでしょう？

デモは[こちらから](#)ダウンロードできます。

結論

この次のチュートリアルをどのように進めるか、かなり困っていました。2つのパートに分けたのは良い解決策だったと思います（希望！）。

ここでは、グラフィックの概念を中心に多くのことを学びました。基本的なVGAの概念、文字や文字列の印刷、画面のクリア、ハードウェアカーソルの更新などについて説明しました。印刷する文字の**属性バイト**を変更することで、簡単に様々な色の文字を印刷することができます。属性バイトの色を変えて、`ClrScr32()`関数を呼べば、新しい背景を作ることもできますよ。カッコいいと思いませんか？退屈な白と黒の世界に風穴をあけることができますね。)

次のチュートリアルでは、ステージ2が終了し、基本的なビュアバイナリの32ビットカーネルイメージを1MBでロードして実行します。心配しないでください。このシリーズのカーネルのセクションに入ると、カーネルのビルド方法を変更し、ロード方法も変更していきます。そうすれば、カーネルをオブジェクト形式でロードして、シンボルのインポートやエクスポートができるようになり、C言語と混ぜることができるようになります。

次のチュートリアルは、新しいことを学ぶという意味でのチュートリアルではありません。その代わり、すでに説明されているコードをすべてカバーしています。しかし、このコードは、コードのレイアウトをより良くするために修正されており、基本的なファイルシステム（FAT12）ドライバとフロッピー・ドライバの間のインターフェース（および分離）を提供しています。とはいえ、これはステージ2の最後のチュートリアルです。

Stage 2は、より多くのオプションを提供したり、**マルチブート**や**ブートオプション**をサポートするように変更することができるので、後ほどStage 2に戻ってみましょう。そのうちわかると思いますが…。)

次の機会まで。

マイク
BrokenThorn Entertainment社。現在、DoEとNeptuneOperating Systemを開発中です。質問や

コメントはありますか？お気軽にお問い合わせください。

あなたも記事の改善に貢献したいと思いませんか？もしそうなら、ぜひ私に教えてください。



第9章

ホーム



第11章



オペレーティングシステム開発シリーズ

オペレーティングシステム開発 - カーネルの準備 part 2

by Mike, 2008

このシリーズは、オペレーティングシステムの開発を一から実演し、教えることを目的としています。

はじめに

歓迎します。:)

前回のチュートリアルでは、プロテクトモードでの基本的なVGAプログラミングについて説明しましたが、さらに1337のデモも作ってみました。

これは、皆さんが待ち望んでいたチュートリアルです。このチュートリアルは、これまでのすべてのコードの上に直接構築されており、1MBの地点でカーネルをロードし、カーネルを実行します。

カーネルはOSの中で最も重要な部分です。カーネル...この謎の敵については、以前にも少しお話ししましたよね？これからのチュートリアルでは、デザイン、構造、開発など、カーネルについてもっと詳しく説明していきます。

今はもう、すべての設定が終わっていますが...。いよいよカーネルをロードして、ステージ2に別れを告げる時が来ました。

注：このチュートリアルは、**Bootloaders 3と4**のチュートリアルの基本的な理解を必要とします。ここではすべてを詳しく説明していますが、すべての概念は**Bootloaders 3および4**のチュートリアルで詳しく説明されています。これらのチュートリアルを読んでいない場合は、まずそれらのチュートリアルをご覧ください。

[OS開発シリーズチュートリアル5:ブートローダ3 OS](#)

[開発シリーズチュートリアル6:ブートローダ4](#)

これらを読んでいれば、このチュートリアルはそれほど難しくありません。

いいですか？

基本的なカーネルスタブ

これが、これから読み込むカーネルです。

```
; 私たちはまだ純粋なバイナリです。次の数回のチュートリアルでこれを修正します :)

オーガ      0x10000          カーネルは1MBでスタート
ビット      32              32ビットコード

jmp         ステージ3        ステージ3へのジャンプ

#include      "stdio.inc"; 前のチュートリアルで作成したstdio.incファイル msg db

0x0A, 0x0A, "Welcome to Kernel Land!!!", 0x0A, 0

Stage3です。

;-----;
; レジスタ                      の設定
;-----;

mov         ax, 0x10          データセクタ (0x10) へのデータセグメントの設定
mov         ds, ax
mov         ss, ax
mov         es, ax
mov         esp, 90000h       スタックは90000hから始まる
abl
;e-----;

; 画面
; をクリアして
; 成功を表示する。
call        _ClrScr32
mov         ebx, msg
call        Puts32

;-----;

;
; 実行
; を停止する
;
;-----;
```

ク
ラ
ス

さて、ここでは特に何もありません。次のセクションでは、このプログラムを大きく発展させていきます。

すべて32ビットであることに注目してください。いいでしょう？ここで完全に16ビットの世界から抜け出すこ

とになります。とりあえず、カーネルにたどり着いたら、システムを停止します。

なお、残りのシリーズでは、このファイルはおそらく一切使用しませんのでご注意ください。むしろ、32 ビットの C++ コンパイラを使うことになるでしょう。カーネルイメージをメモリにロードした後、メモリ内のファイルを解析してカーネルエントリルーチンを探し、Cのmain()ルーチンを第2段階ブートローダから直接呼び出すことができます。かっこいいでしょう？つまり、第2段階のブートローダから、スタブファイルやプログラムなしで、C++の世界に直接入ることができるのです。しかし、出発点が必要です。そこで、このチュートリアルでは、基本的なスタブファイルを使って、動作を確認してみましょう。

次の数回のチュートリアルでは、コンピリアを立ち上げて動作させ、その代わりに使用します。しかし、ここでは先の話をしています。)

フロッピーインターフェース

イエーイステージ2を終了する時が来ました。カーネルをロードするためには、再びFAT12を横断する必要があります。しかし、その前にディスクからセクタを取り出す必要があります。

このコードはブートローダと全く同じで、BIOSのINT 0x13を使ってディスクからセクタをロードします。

このチュートリアルは完全な復習でもあるので、各ルーティンをセクションに分けて、何が起きているのかを正確に説明します。

セクターの読み込み - BIOS INT 0x13

ブートローダ3では、セクタの読み込みに必要なものについて説明しました。チュートリアルを振り返って、BIOS割り込み0x13の機能2を使ってセクタを読み込めることを思い出してください。さて、それでは。ここで問題なのは、プロテクトモードに入る前にセクタをロードしなければならないことです。プロテクトモードからBIOS割り込みを呼び出そうとすると、プロセッサがトリプルフォールトになってしまいますよね？

とりあえず、割り込みはどうだったの？右....

INT 0x13/AH=0x02 - DISK : READ SECTOR(S) INTO MEMORY

AH = 0x02

AL = 読み込むセクタの数

CH = シリンダー番号の下位8ビット

CL = セクター番号（ビット0〜5）。ビット6-7はハードディスクのみ

DH = ヘッド・ナンバー

DL = ドライブ番号（ハードディスクの場合はビット7を設定） ES:BX = セクタを読み出すバッファ

戻ります。

AH = ステータスコード

AL = 読み込んだセクタ数

CF = 失敗したらセット、成功したらクリア

これはそれほど難しいことはありません。しかし、ブートローダのチュートリアルを思い出してください。つまり、セクタ、トラック、ヘッド番号を追跡し、トラックを超えるセクタをロードしようとしなくする必要があります。つまり、1つのトラックには18個のセクタがあることを覚えてい

ますか？セクター番号を18より大きくすると、コントローラが故障し、プロセッサがトリプルフォールトになります。

じゃあ、いいですか！？この情報のすべてが。トラックごとのセクタ、トラック数、ヘッド数、セクタのサイズ、完全にディスク自体に依存します。セクターは512バイトである必要はないことを覚えてい

ますか？OEM Parameter Blockにすべてを記述しています。

```
bpbOEM                db "私のOS  "
bpbBytesPerSector      dw 512
bpbSectorsPerCluster   db 1
bpbReservedSectors     dw 1
bpbNumberOfFATs:      db 2
bpbRootEntries:        dw 0x20 ;; 0xF1
bpbTotalSectorsFAT:    dw 3880
bpbSectorsPerTrack:    dw 18
bpbHeadsPerCylinder:   dw 2
bpbHiddenSectors:      dd 0
bpbTotalSectorsBig:    dd 0
bsDriveNumber:         db 0
bsUnused:              db 0
bsExtBootSignature:    db 0x29
bsSerialNumber:        dd 0xa0a1a2a3
bsVolumeLabel:         db "mos floppy"
bsFileSystem:          db "FAT12  "
```

2021/11/15 13:07

オペレーティングシステム開発シリーズ

これは見覚えがあるはずです。各メンバーはチュートリアル5で説明されていますが、ここでのすべての詳細な説明については、そのチュートリアルを参照してください。

あとは、ディスクからメモリのある場所に任意の数のセクタをロードするためのメソッドがあればよいのです。しかし、すぐに問題が発生します。ロードしたいセクタはわかっています。しかし、**BIOS INT 0x13**には

これが何の関係があるのでしょうか？例えば、セクター20をロードしたいとします。**1**トラックには**18**セクターしかないので、この数字を直接使うことはできません。現在のトラックの第**20**セクターから読み込もうとすると、そのセクターが存在しないため、フロッピーコントローラが故障し、プロセッサがトリプルフォールトになります。**20**番目のセクターを読み取るためには、トラック**2**セクター**2**、ヘッド**0**を読み取る必要があります。これは後で検証します。

CHSからLBAへの会話の手順を覚えていますか？

これは聞き覚えがあるのではないのでしょうか？ **LBA (Linear Block Addressing)** とは、簡単に言えば、ディスク上のインデックス付きの場所を表すものです。最初のブロックは0、2番目のブロックは1です。言い換えれば、LBAは単純に0から始まるセクター番号を表し、各「ブロック」は1つの「セクター」となります。

読者の方の中には、このコードがかなりトリッキーだとおっしゃる方がいました。そこで、ここではその詳細を説明したいと思います。まず、`formulas`をもう一度見てみましょう。

これを例に挙げてみましょう。第20セクターはトラック2、セクター2であるべきだと言いましたよね？では、この式を実際に試してみましょう。

[illegible]

さて、このルーチンは1つのパラメータを取ります。CHSに変換する論理セクタを含むAXです。式に注意してください（**論理セクタ / トラックあたりのセクタ数**）は、3つの計算式に含まれています。これを何度も計算し直すよりも、単に計算の方が効率的です。その結果を他のすべての計算に使用する...。これがこのルーチンの仕組みです。

セクター1から始める（論理的なセクター／トラックごとのセクターの「+1」を覚えている？

```
inc     dl                      セクター0で調整
.mo     BYTE [absoluteSector], dl
vie
```

DXをクリアします。AXにはトラックごとの論理セクタ/セクタの結果が残っている

```
xor     dx, dx                  dx:axの動作準備
```

さて、数式ですが....。

アブソリュート・ヘッド = (論理セクター / セクタ/トラック) MOD ヘッド数

ブソリュート・トラック = 論理セクター / (セクタ/トラック * ヘッド数)

掛け算の結果は、頭の数だけの**割り算**になります。つまり、この2つの違いは操作方法だけで、1つは割り算、1つはその割り算の余り（モジュラス）なのです。

では、剰余（MOD）と除算の結果の両方を返せる命令は何でしょうか？DIV!

論理セクタ数 / トラックあたりのセクタ数)はAXのままなので、あとはシリンダーあたりのヘッド数で割るだけです。

```
div     WORD [bpbHeadsPerCylinder]    計算
        (英語)
```

アブソリュート・ヘッドとアブソリュート・トラックの方程式は非常によく似ています。実際の違いは操作方法だけです。この単純なDIV命令は、**DXとAXの両方を設定します。AXにはHeadsPerCylinderのDIVISIONが格納され、DXには同じ演算のREMAINDER(Modulus)が格納されます。**)

```
mov     BYTE [absoluteHead], dl
mov     BYTE [absoluteTrack], al
ret
```

これで少しはすっきりしたでしょうか。もしそうでなければ、ぜひ教えてください；)

CHSからLBAへの変換

これはもっとシンプルなものですよ。

```
ClusterLBAです。
        L          (クラスタ - 2 ) * クラスタあたりのセクタ
BA=      数
サブ     ax, 0x0002          クラスタナンバーから2を引く
xor      cx, cx
movab    cl, BYTE [bpbSectorsPerCluster]    クラスタあたりのセクタ数
le       マルチ    cx          マルチプライ
```

セクターで読む

さて、これですべてのセクタを読み取ることができました。このコードもブートローダからのものと全く同じです。

```
;*****;
一連のセクタを読む
CX=>読み取るセクタの数
; AX=>開始セクター
; ES:BX=>読み込み先のバッファ
;*****;

ReadSectors
    です。
.MAIN     ディ,          エラー時には5回のリトライ
movable   0x0005
```

さて、ここではセクタを5回読むを試みます。

```
.SECTORLOOP
    プッシュ   軸
    ユ         bx
    プッシュ   bx
    ユ         cx
    プッシュ   cx
    ユ         LBACHS          スタートセクターをCHSに変換
    コール
```


2021/11/15 13:07

オペレーティングシステム開発シリーズ

レジスターをスタックに格納します。開始セクタはリニアセクタ番号です（AXに格納されています）。BIOSのINT 0x13を使用しているので、ディスクから読み込む前にCHSに変換する必要があります。そこで、LBAからCHSへのカバーションルーチンを使用します。**absoluteTrack**にはトラック番号が、**absoluteSector**にはトラック内のセクタが、**absoluteHead**にはヘッド番号が入ります。これらはすべて、LBA→CHAの会話ルーチンで設定したものですよね？

movab	ああ、0x02	ズ	; BIOSリードセクター
le			
movab	al, 0x01		1セクタの読み取り
le			
movab	ch, BYTE [absoluteTrack]		トラック
le			
movab	cl, BYTE [absoluteSector].		セクター
le			
movab	dh, BYTE [absoluteHead] (アブソ		ヘッド
le	リュートヘッド		
movab	dl, BYTE [bsDriveNumber		ドライブ
le			
int	0x13		BIOSを起動する

これで、セクタを読み取るための準備が整い、BIOSに読み取らせることができました。簡単にするために、実行しているBIOSのINT 0x13ルーチンをもう一度見てみましょう。

INT 0x13/AH=0x02 - DISK : READ SECTOR(S) INTO MEMORY

AH = 0x02
AL = 読み込むセクタの数
CH = シリンダー番号の下位8ビット
CL = セクター番号 (ビット0〜5)。ビット6-7はハードディスクのみ
DH = ヘッド・ナンバー
DL = ドライブ番号 (ハードディスクの場合はビット7を設定) ES:BX = セクタを読み出すバッファ
ア

上のコードの実行方法と比較してみてください、とてもシンプルでしょう？

書くべきバッファはES:BXにあり、INT 0x13がバッファとして参照されることを覚えておいてください。ES:BXをこのルーチンに渡しているので、これがセクタをロードする場所になります。

JNC	.SUCCESS	読み込みエラーのテスト
xor	ax, ax	BIOSリセットディスク
int	0x13	BIOSを起動する
dec	ディ	エラーカウンタのデクリメント
ポップ	cx	
ポップ	bx	
ポップ	軸	
jnz	.SECTORLOOP	再読の試み

BIOSのINT0x13関数2は、エラーがある場合にキャリーフラグ (CF) を設定します。エラーがあった場合は、カウンタをデクリメントして (ループを5回試行するように設定したのを覚えていますか？

5回のアッテムがすべて失敗した場合 (CX=0、ゼロフラグセット) 、INT 0x18命令にフォールダウンします。

```
int    0x18
```

...コンピュータを再起動します。

キャリーフラグがセットされていない (CF=0) 場合、エラーがなかったことを示すため、jnz命令はここでジャンプします。セクターは正常に読み込まれました。

.SUCCESS		
ポップ	cx	
ポップ	bx	
ポップ	軸	
追加	bx, WORD [bpbBytesPerSector]を使	次のバッファをキューに
	用します。	入れる
inc	軸	次のセクターをキューに
		入れる
ループ	.MAIN	次のセクタを読む
リセット		

あとは、レジスターを復元して、次のセクターに行くだけです。難しいことはありません)なお、ES:BXにはセクタをロードするアドレスが含まれているので、次のセクタに行くにはBXをセクタあたりのバイト数だけインクリメントする必要があります。

AXには読み出すための開始セクタが含まれていたの、これもインクリメントする必要があります。

とりあえず、これで終わりです。このルーチンの詳しい説明については、Bootloaders 4 を参照してください。

フロッピー16.inc

デモの例では、フロッピーアクセスのルーチンはすべてFloppy16.incの中にあります。

FAT12インタフェース

セクターの読み込みができるようになりました。うわあ... :(ご存知のように、これでは実際にはあまり何もできません。次にやるべきことは、「ファイル」の基本的な定義と「ファイル」とは何かを作ることです。これには、ファイルシステムを使います。

ファイルシステムは非常に複雑です。このコードがどのように動作するかを完全に理解するには、**Bootloaders 4**を参照しながらこのコードを説明してください。

定数

Fat12の解析では、ルート・ディレクトリ・テーブルとFATテーブルを読み込むための場所が必要になります。これを簡単にするために、これらの場所を定数で隠します。

```
%define ROOT_OFFSET 0x2e00
%define FAT_SEG 0x2c0
%define ROOT_SEG 0x2e0
```

ここでは、ルート・ディレクトリ・テーブルを0x2e00に、FATを0x2c00にロードします。FAT_SEGとROOT_SEGはセグメント・レジスタへのロードに使用します。

FAT12のトラバース

ご存知のように、OSのコードの中には単純に醜いものがあります。私の意見では、ファイルシステムのコードもその一つです。これが、このチュートリアルでFAT12のコードを説明することにした理由の1つです。FAT12のコードは基本的にブートローダと同じですが、メインプログラムとの依存関係を減らすために修正することにした。このため、ここでは詳細に説明することにした。

ここでは、FAT12の詳細については説明しません。詳細はBootloaders 4のチュートリアルを参照してください。

さて、ご存知のように、FAT12を横断するためには、まず、Root Directory Tableを読み込む必要があります。

ルートディレクトリテーブルの読み込み

ディスクの構造。

ブート セクター	エクストラ ードセク ション	ファイルア ロケーショ ンテー ブル1	ファイルア ロケーショ ンテー ブル2	ルートディ レクトリ (FAT12/FAT16のみ)	ファイル やディレ クトリを 含むデー タ領域。
-------------	----------------------	------------------------------	------------------------------	----------------------------------	--------------------------------------

Root Directory TableがFATやReserved Sectorのすぐ後にあることを覚えていますか？

ルートディレクトリテーブルの読み込みでは、メモリ上で現在必要のない場所を探し、そこにコピーする必要があります。ここでは、0x7E00（リアルモード：0x7E0:0）を選択しました。これはブートローダのすぐ上にあります。ブートローダは一度も上書きされていないので、まだメモリ内にあります。

ここには重要なコンセプトがあります。すべてを絶対的なメモリ位置にロードしなければならないことに注意してください。これは、どこに何があるのかを物理的に把握しなければならないため、非常に悪いことです。そこで登場するのが低レベルのメモリマネージャです。詳しくは後ほど...

```
;*****
; LoadRoot ()
;   - ルートディレクトリテーブルの読み込み
;*****

LoadRoot:

    ; ブシャ          レジスタの保存
    ;   ・プッ      es
    ;   シュ
```

まず、レジスタの現在の状態を保存します。これをしないと、それを使用するプログラムの残りの部分に影響を与えることになり、非常にまずいことになります。次に、ルートディレクトリテーブルのサイズを取得して、ロードするセクタの数を知ります。

Remember from Bootloaders 4: 各エントリは32バイトのサイズです。FAT12フォーマットのディスクに新しいファイルを追加すると、Windowsは自動的にルート・ディレクトリに追加し、OEMパラメータ・ブロックのbpbRootEntriesバイト・オフセット変数に追加します。

ほら、Windowsっていいじゃないですか！)

つまり、各エントリのサイズが32バイトであることがわかっているの、32バイトにルートディレクトリの数を掛けると、Root Directory Tableに何バイトあるかがわかります。単純なことです、セクタ数が必要なので、この結果をセクタ数で割る必要があります。

```

; ルートディレクトリのサイズを計算し、"cx "に格納する。
xor     cx, cx          ; レジスタのクリア
xor     dx, dx          ;
movabl  アックス, 32    ; 32バイトのディレクトリエントリ
e       ;
マルチ WORD 「bpbRootEntries ; ディレクトリの合計サイズ
div     WORD [bpbBytesPerSector]を使用し ; ディレクトリが使用するセクタ
        ます。
xchg    ax, cx          ; AXへの移行
```

さて、これでAX=ルートディレクトリが取るセクタの数となりました。さて、次はスタート地点を見つけなければなりません。

ブートローダ4の覚え書き：ルートディレクトリテーブルは、ディスク上のFATと予約セクタの両方のRight afterです。上記のディスク構造表を見て、ルート・ディレクトリ・テーブルがどこにあるかを確認してください。

つまり、FATのセクタ数を求め、それを予約セクタに加えて、ディスク上の正確な位置を求めればいいのです。

```

; ルートディレクトリの位置を計算し、"ax "に格納する。
movab   al, byte [bpbNumberOfFATs]. ; FATの数
le      マルチ   ワード「bpbSectorsPerFAT ; FATが使用するセクタ
```

追加 movab le 追加	アクセス、ワード【bpbReservedSectors ワード[データセクタ]、アクセス ワード [データセクタ], cx	ブートセクタの調整 ルートディレクトリのベース
-------------------------	---	----------------------------

読み込むセクタ数と正確な開始セクタがわかったところで、読み込んでみましょう。

ルートディレクトリの読み込み		
プッシュ	ワード ROOT_SEG	
ユポップ	es	
movab1	bx, 0x0	コピー・ルート・ディレクトリ
eコール	ReadSectors	ディレクトリ・テーブルに読み込まれる
ポップ	es	
ボバ		レジスタを復元して戻る
レット		

ROOT_SEG:0に読み込まれるようにseg:offsetの位置を設定したことに注

目してください。次はFATの読み込みです！

FATの読み込み

さて、**Bootloaders 4**では、FAT12フォーマットのディスク構造についてお話ししましたね。今回は過去にさかのぼって、もう一度見てみましょう。

ディスクの構造。

ブート セクタ ー	エクストラリザ ードセクショ ン	ファイルアロケ ーションテーブ ル1	ファイルアロケ ーションテーブ ル2	ルートディレクトリ (FAT12/FAT16のみ)	ファイルやディレクトリを含むデー タ領域。
-----------------	------------------------	--------------------------	--------------------------	------------------------------	--------------------------

FATは1つまたは2つあることを覚えていますか？また、ディスク上の予約済みセクタの直後にあることにも注目してください。これは見覚えがあるはずですよ。

***** ; LoadFAT () ; ; - FATテーブルのロー ; ; ド ; Parm/ ES:DI => ルートディレクトリテーブ ; ル *****		
LoadFATです。		
ブシャ ・プッ シユ	es	レジスタの保存

まず、ロードするセクタ数を知る必要があります。もう一度、ディスク構造を見てみましょう。OEMパラメータ・ブロックにFATの数(およびFATあたりのセクタ数)を格納しています。ですから、総セクタ数を得るには、それらを乗算すればよいのです。

FATのサイズを計算し、"cx "に格納する。		
xor movab le マルチ movab le	ax, ax al, BYTE [bpbNumberOfFATs]. ワード「bpbSectorsPerFAT」 cx, ax	FATの数 FATが使用するセクタ

さて、予約済みのセクタを、FATの前と同じようにコインロッカーに入れる必要があります…。

FATの位置を計算し、"ax "に格納する。		
mov abl e	アクセス、ワード【 bpbReservedSectors	

ようしゃー！CXにはロードするセクタ数が入っているので、セクタをロードするためのルーチン呼び出して下さい。

それが全てでFATをメモリに読み込む(0x7c00のブートルードを上書きする)

ファイルの検索

pushword ポップス xorbx, bx はReadSectorsのポ ップスを呼び出します	FAT_SEG popa; レジスタをリストアしてret
---	---------------------------------

ファイルを検索するには、検索対象となるファイル名が必要です。DOSでは、一般的な8.3命名規則（8バイトのファイル名、3文字の拡張子）に従った11バイトのファイル名を使用することを覚えておいてください。ルートディレクトリのエントリの構造上、これは11バイトでなければなりません。

Root Directory Tableのフォーマットを覚えておきましょう。ファイル名は、エントリの最初の11バイトに格納されます。それでは、各ディレクトリエントリのフォーマットをもう一度見てみましょう。

- バイト **0-7** :DOSファイル名（スペースでパッドされている） **8**
- ~10バイト目DOSファイルの拡張子（スペースでパッドされてい
- る） バイト**11** :ファイルの属性。これはビットパターンです。
 - ビット**0** :Read
 - Only **Bit 1** :
 - Hidden **Bit 2** : System
 - **Bit 3** : ボリュームラベル
 - ビット**4** : これはサブディレクトリです
 - **Bit 5** : アーカイブ
 - ビット**6** : デバイス（内部使用
 - ビット**6** : 未使用
- **Bytes 12** :未使用
- **Bytes 13** :作成時間（単位：ms
- **14~15**バイト目：作成時刻（以下のフォーマットで表示
 - ビット**0-4** : 秒数（0-29
 - ビット **5-10** :分（0~59
 - **Bit 11-15** : 時間(0-23)
- **16byte-17byte** : 作成した年を以下の形式で表示します。
 - ビット**0-4** : 年（0=1980、127=2107
 - ビット**5-8** :月（1=1月、12=12月）
 - ビット**9-15** : 時間(0-23)
- Bytes **18-19** :Bytes **20-21** : EAインデックス（OS/2やNTで使用されて
- いますが、気にしないでください） **Bytes 22-23** : Last Modified time（
- フォーマットはバイト14-15を参照してください。Bytes 22-23 : Last
- Modified time（フォーマットはbyte 14-15参照） **Bytes 24-25** : Last
- modified date（フォーマットはbyte 16-17参照） **Bytes 26-27** : First
- **Cluster（クラスタ）ファーストクラスタ**
- **Bytes 28-32** : ファイルサイズ

太字のエントリはすべて重要なエントリです。各エントリの最初の11バイトにはファイル名が含まれているので、これを比較する必要があります。

一致するものが見つかったら、そのエントリーのバイト26を参照して、現在のクラスタを取得する必要があります。ここまではおなじみですね。

では、コードを見てみましょう。

```
;*****
検索ファイル ()。
    ; - ファイル名をルートテーブルで検索
;
; parm/ DS:SI => ファイル名
; ret/ AX => ディレクトリテーブルのファイルインデックス番号。エラーの場合は-1
;*****

FindFile:
    push    cx                    レジスタの保存
    push    dx
    push    bx
    movabl  bx, si                ファイル名を後に残す
    e
```

まず、現在のレジスタの状態を保存します。SIを使う必要があるので、現在のファイル名をどこかに保存する必要があります。

イメージ名を見つけるためには、Root Directoryテーブルを解析する必要があることを覚えておいてください。そのためには、ディレクトリテーブルの各エントリの最初の11バイトをチェックして、一致するものがあるかどうかを確認する必要があります。簡単そうでしょうか？

そのためには、エントリーの数を知る必要があります。

バイナリイメージのルートディレクトリの参照

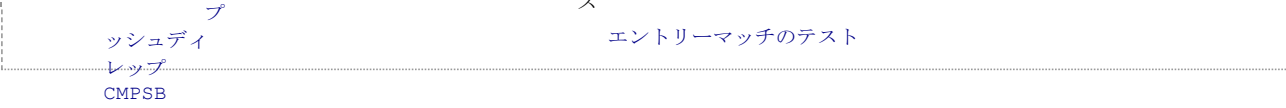
```
movab    cx, word [bpbRootEntries].    ループカウンタのロード
le
movab    di, ROOT_OFFSET                最初のルートエントリーを探す
le
cld                                       クリアディレクションフラグ
```

さて、これでCXには調べるべきエントリーの数が入りました。あとはループして11バイトの文字列ファイル名を比較するだけです。ここでは文字列命令を使用しているので、まず方向フラグがクリアされていることを確認します。

DIは、ディレクトリテーブルの現在のオフセットに設定されます。つまり、ES:DIはテーブルの開始位置を指しているの、これを解析してみましょう。

```
.LOOPです。
    プッシュ      cx
    movable      cx, 11
    movable      si, bx
```

11文字の名前があります。画像名はSI
画像名がBXの場合



11バイトが一致すれば、そのファイルは見つかったことになる。DIはテーブル内のエントリの位置を含んでいるので、すぐに
.見つかりました。

一致しない場合は、テーブルの次のエントリを試してみる必要があります。DIに**32**バイト追加します。(各エントリが**32**バイトであることを覚えて
いますか?)

pop
je
pop
add
loop

ディ
.Found
cx
ディ、
32
.LOOP

キューの次のディレクトリエントリ

ファイルが見つからない場合は、スタックに残っているレジスタのみをリストアし、-1（エラー）を返します。

.NotFound
です。
ポップ
ポップ
ポップ
movab
le
レット

bx
dx
cx
軸、-1

レジスターを復元して戻る

セットのエラーコード

ファイルが見つかった場合は、すべてのレジスタを復元します。AXには、Root Directory Table内のエントリ位置が格納されているので、それを読
み込むことができます。

.見つかりました
o
ポップ
ポップ
ポップ
ポップ
レット

軸
bx
dx
cx

AXへの戻り値は、ファイルのエントリを含んでいます。
レジスターを復元して戻る

やったーこれで、ファイルを見つけることができました（Root Directory Tableの中での位置もわかりました）。

ファイルの読み込み

全ての設定が完了したので、いよいよファイルを読み込みます。

この部分のほとんどは、他のルーチンと呼び出しているもので、とても簡単です。ここではループして、ファイルのクラスタがすべてメモリに読み込
まれていることを確認します。

;*****
; LoadFile ()
; - ファイルの読み込み
; parm/ ES:SI => ロードするファイル
; parm/ BX:BP => ファイルを読み込むためのバッファ
ret/ AX => エラー時に-1、成功時に0
; ret/ CX => ロードされたセクタの数
;*****

LoadFileです。
xor
push

ecx, ecx
ecx

ここではレジスタを保存するだけです。どこかに書き込むためにバッファのコピーを保持する必要があるので、それもスタックに保存します。CX
は、いくつかのセクタをロードしたかを追跡するために使用されます。これは後で使えるようにスタックに保存しておきます。

ファイルをロードするには、まずファイルを見つける必要があります（当たり前ですよ^^）ここでは、FindFileルーチンを簡単に使うことがで
きます。FindFileは、エラー時にはAXを-1に設定し、**成功時にはRoot Directory Table内の開始エントリ位置**を設定します。このインデックス
を使って、そのファイルについて知りたいことを何でも知ることができます。

.FIND_FILE
です。

プッシュ
ユ
プッシュ
ユ

コール

bx
bp

ファイル検索

; BX=>BPは、書き込み先のバッファを指し、後で保存する

は、私たちのファイルを見つけます。ES:SIはファイル名を含む


```
cmp    軸、-1
jne    .load_image_pre
pop    bp
push   bx
pop    ecx
push   bp
push   bp
pop    bp
push   bp
```

```
エラーのチェック
エラーはありません。)FATの読み込み
; Nope :( レジスタを復元し、エラーコードを設定して戻る
```

```
mov     軸、-1
ret
reat
t
```

さて、ここまでくれば、ファイルは見つかったということになります。**ES:DI**には、**FindFile()**によって設定された最初のルートエントリの場所が含まれているので、**ES:DI**を参照することで、実質的にファイルのエントリを取得することになります。

前節の上のエントリ記述表を見てください。0x1Aバイトのオフセットでバイト26（開始クラスタ番号）に到達できることに注目して、それを格納して...

```
.load_image_pre:

    サブディ、
    ROOT_OFFSET
    subeax,
    ROOT_OFFSET

    クラスタを起動する
    pop     ROOT_SEG es
    mov     dx, word [es:di + 0x001A]; ES:DIは、ルートディレクトリテーブルのファイルエントリを指し
    pop     ます。ド[クラスタ]、      ; ファイルの最初のクラスタのテーブルを再表示します。
    pop     dx bx                      スタックを崩さないように書き込み先を確保する。
    push    es
    push    bx                      後日のための保存場所
    push    es
```

上記は面倒くさいですね。AXにはFindFileの呼び出しによってエントリ番号が設定されたことを覚えていますか？これをここに保存する必要がありますが、書き込み用のバッファはスタックの一番上に置いておく必要があります。これが、ここでスタックを少し弄った理由です :)
とにかく、次はFATを読み込みます。これは信じられないほど簡単です...

```
        コールロ                      0x7c00にFATをロードする。
        ードファット
```

OKです。これでFATが読み込まれ、開始ファイルのクラスタができたので、実際にファイルのセクタを読み込むことになります。

```
.LOAD_IMAGE
です。
mov     ax, WORD [cluster]          読みたいクラスタ
pop     es
pop     bx
call    ClusterLBA                  クラスタをLBAに変換
xor     cx, cx
mov     cl, BYTE [bpbSectorsPerCluster] ; 読み込むセクター
call    ReadSectors                 クラスタで読む

    ポップ      ecx                  セクター数の増加
    プ          ecx
    イン        ecx

    ク          bx                  次のイテレーションのためにレジスターを保存
    プッシュ    es
    シュ        ax
    ュ          FAT_SEG
    ュ          es, ax
    ュ          bx, bx
```

このコードはそれほど悪いものではありません。FAT12では、各クラスタはちょうど**512バイト**であることを覚えています。つまり、各クラスタは単に「セクタ」を表します。まず最初にクラスタ/セクタの開始番号を取得します。クラスタ番号は**線形**なので、クラスタ番号だけではあまり意味がありません。つまり、これは**CHS Not LBA**フォーマットのセクタ番号であり、トラックとヘッ드의情報を持っていることを前提としています。ReadSectors()ではLBAのリニアセクタ番号が必要なので、この**CHSをLBAアドレスに変換**します。そして、クラスタごとのセクタを取得して、それを読み込みます。

ESとBXをポップすることに注意してください--これらは最初からスタックにプッシュされています。**ES:BX**は、このルーチンに渡された**ES:BP**バッファを指しています--これには、セクタをロードするためのバッファが含まれています。

さて、クラスタが読み込まれたところで、FATでファイルの終端に達しているかどうかを確認しなければなりません。しかし。**FATの各エントリが12バイトであることを覚えていますか？ブートローダ4から、FATを読むときにパターンがあることがわかりました。**

偶数クラスタの場合は低位12ビット、高位クラスタの場合は高位12ビットを取得します。

詳しくは「ブートローダ4」をご覧ください。偶数か

奇数かを判断するには、2で割るだけです。

```
        次のクラスタを計算する

        movab   ax, WORD [cluster].      現在のクラスタを特定する
        le
```

```
movab cx, ax ; 現在のクラスタをコピー
le
movab dx, ax ; 現在のクラスタをコピー
le
shr dx, 0x0001 2で割る
追加 cx, dx ; の合計 (3/2)

movabl bx, 0 ; メモリ内のFATの位置
e
追加 bx, cx ; FATへのインデックス
movabl dx, WORD [es:bx] で ; FATから2バイトを読み込む
e
テスト ax, 0x0001
jnz .ODD_CLUSTER

.EVEN_CLUSTER: 0000111111111111b ; 低い12ビットを取る
として ax,
jmp .DONE

.ODD_CLUSTER:
shr dx, 0x0004 ; テイクハイ12ビット

.DONE
movab WORD [クラスタ], dx ; 新しいクラスタの保存
le
cmp dx, 0xFF0 ; ファイル終了マーカ (0xFF) のテスト
jb LOAD_IMAGE ; いいえ？次のクラスタに進んでください。

DONE。
ポップ es ; 全レジスタの復元
pop bx
pop ecx
xor ax, ax ; 成功コードを返す
レット
```

これが全てです。ちょっと複雑だけど、難しくはないと思います。)

Fat12.inc

素晴らしいですね。FAT12のコードはすべてFat12.incにあります。

フィニッシングステージ2

ステージ2に戻る - カーネルのロードと実行

煩雑なコードが終わったので、あとはステージ2からカーネルイメージをメモリにロードして、カーネルを実行するだけです。問題はどこで？

1MBにロードしたいとは思っていますが、まだ直接はできません。その理由は、まだリアルモードだからです。このため、まず、イメージをより低いアドレスにロードする必要があります。プロテクトモードに切り替えた後、カーネルを新しい場所にコピーします。これは1MBでも、ページングが有効であれば3GBでもよい。

```
コール LoadRoot ; ルートディレクトリテーブルの読み込み

movabl ebx, 0 ; BX:BPはロード先のバッファを指します。
e
movabl ebp, IMAGE_RMODE_BASE
e
movabl Esi, ImageName ; ロードするファイル
e
コール ロードファイル ; ファイルの読み込み
MOV dword [ImageSize], ecx ; カーネルのサイズ
cmp ax, 0 ; 成功のためのテスト
jee EnterStage3 ; そう、Stage3へ。
movabl si, msgFailure ; Nope--プリントエラー
e
コール Puts16
movabl あ, 0
e
int 0x16 ; await keypress
int 0x19 ; ウォームブートコンピュータ
クライ ; ここまで来ると、何かが間違っていたとしか思えません
アント
halt
```

これで、カーネルがIMAGE_RMODE_BASE:0にロードされました。ImageSizeには、ロードされたセクタ数（カーネルのサイズ）が入ります。

プロテクトモード内で実行するために必要なのは、ジャンプまたはコールすることだけです。カーネルを1MBにしたいので、実行する前にまずカーネルをコピーする必要があります。

ビット
32
Stage3
です。

```
mov    AX, DATA_DESC
mov    ds, ax
mov    ss, ax
mov    es, ax
mov    esp, 90000h
```

データセクタ (0x10) へのデータセグメントの設定

スタックは90000hから始まる

```
; カーネルを1MBにコピー (0x10000)
```

CopyImageです。

```
        moveax, dword [ImageSize] (イメージサイズ)
movzx   ebx, word [bpbBytesPerSector].
```

```

mul     ebx
mov     ebx, 4
div     ebx
cld
mov     esi, IMAGE_RMODE_BASE
mov     edi, IMAGE_PMODE_BASE
rep     ecx, eax
movsd

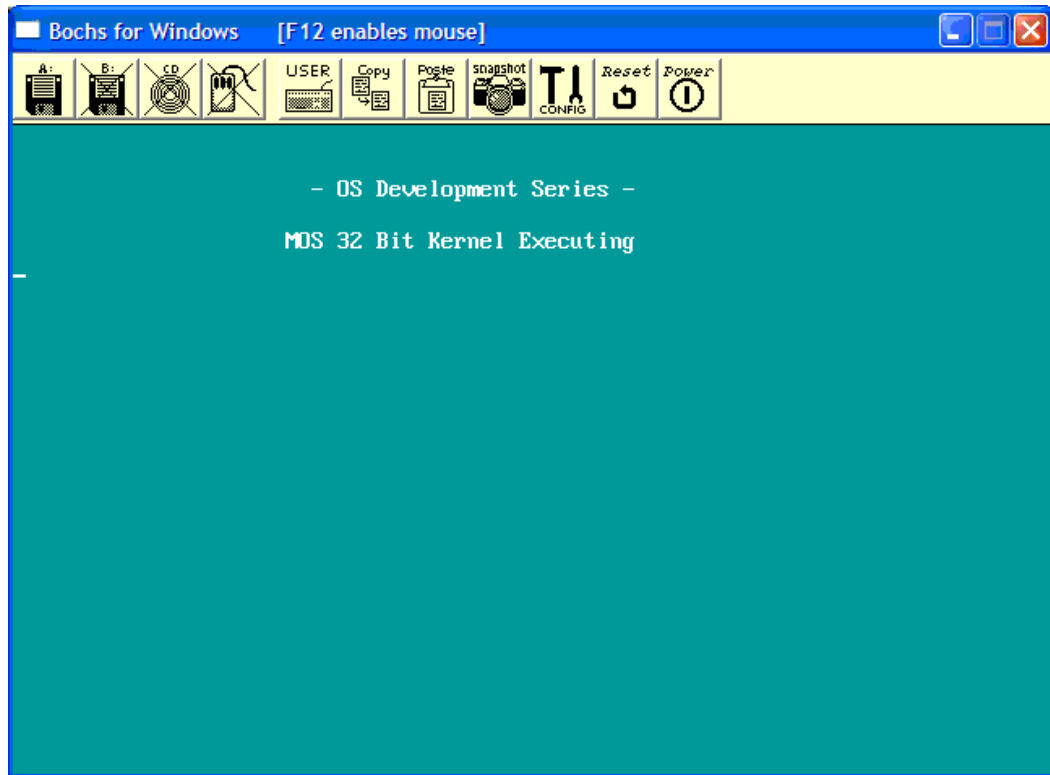
```

コピーイメージをプロテクトモードのアドレスに

CODE_DESC:IMAGE_PMODE_BASEを呼び出して、カーネルを実行します。

しかし、ここにはちょっとした問題があります。これは、カーネルが**純粋なバイナリファイル**であることを前提としています。**C**はこれをサポートしていないので、これではいけません。カーネルは、**C**がサポートするバイナリ形式である必要があります。そして、カーネルをC.今のところ、この純粋なバイナリを維持しますが、次の数回のチュートリアルでこれを修正します。かっこいいと思いませんか？

デモ



私たちの純粋なuber-1337 32bit Kernel

executing. [デモのダウンロードはこちら](#)

結論

W00t!カーネルタイムだ!!!:)

このチュートリアルでは、多くの新しいコードを取り上げました。このチュートリアルのコンセプトのほとんどは、以前にも説明したことがあるので、このチュートリアルがそれほど難しくなかったことを願っています。)

しかし、このチュートリアルでは、これらの概念を新しい視点でカバーしています。このチュートリアルでは、これらの概念を新しい視点で説明しています。これにより、これらのトピックをより深く理解し、それぞれのルーチンに組み込むことができるようになります。

ディスクからセクターをロードし、**FAT12**を解析してカーネルを好きな場所にロードするコードを開発しました。すごいでしょ？このシリーズでは、カーネルを**1MB**にロードします。

基本的なフル**32ビット**カーネルがようやくロードされて実行されるようになったので、ようやくオペレーティングシステムの最も重要な部分であるカーネルに注目することができるようになりました。

次の数回のチュートリアルでは、**Kernel Theory, Revolutions, and Designs**を取り上げます。その後、**低レベルCプログラミング**、および高レベル言語の概念と理論を用いた低レベルプログラミングを取り上げます。

C言語をカーネルレベルでプログラミングすると、他の多くのプログラミング分野では許されない自由度があります。例えば、「アクセス違反」というものがないので、メモリ上のすべてのバイトを直接制御することができます。悪いニュースがあります。また、「標準ライブラリ」というものはありません。さらに悪いニュースとしては、自分は**低レベルの環境**をプログラミングしているのであって、**C**言語という別の抽象化されたレイヤーを使っているだけだということを忘れてはいけません。

次の数回のチュートリアルでは、すべてをカバーし、カーネルで動作するように**C**言語を設定する予定です。待ち遠しいですね。

では、お会いしましょう。 ;)

マイク

BrokenThorn Entertainment社。現在、DoEと[Neptune Operating System](#)を開発中です。質問

やコメントはありますか？お気軽に[お問い合わせください](#)。

あなたも記事の改善に貢献したいと思いませんか？もしそうなら、ぜひ[私に教えてください](#)。



第10章

ホーム

第12





オペレーティングシステム開発シリーズ

オペレーティングシステム開発 - カーネル。基礎知識編1

by Mike, 2009

このシリーズは、オペレーティングシステムの開発を一から実演し、教えることを目的としています。

はじめに

歓迎します。:)

さて...私たちはついに、あらゆるオペレーティングシステムの最も重要な部分にたどり着きました。**カーネル**です。

この言葉は、これまでのシリーズで何度も耳にしてきました。その理由は、この言葉がいかに重要であるかということです。

カーネルは、すべてのオペレーティングシステムの中核をなすものです。カーネルとは何か、そしてそれがオペレーティングシステムにどのような影響を与えるのかを理解することが重要です。

このチュートリアルでは、カーネルの背後にあるもの、カーネルとは何か、カーネルは何を担当しているのかを見ていきます。これらの概念を理解することは、良いデザインを考える上で必要不可欠です。

いいですか？

カーネル。基本的な定義

OSの**カーネル**とは何かを理解するためには、まず「**カーネル**」とは何か、その基本的な定義を理解する必要があります。辞書によると、「カーネル」は「核」、「本質的な部分」、あるいは「何かの本体」と定義されています。この定義をOS環境に当てはめると、次のようなことが簡単に言えます。

カーネルは、オペレーティングシステムの中核となるコンポーネントです。

でも、これは私たちにとってどういう意味があるのでしょうか？OSのカーネルとは一体何なのでしょう？そしてなぜ気にしなければならないのでしょうか？

カーネルが必須であるというルールはありません。カーネル」がなくても、特定のアドレスにプログラムをロードして実行することは簡単にできます。実際、初期のコンピュータシステムはすべてこの方法でスタートしました。最近のシステムでも、これを採用しているものがあります。顕著な例としては、初期の**counsole**ビデオゲームシステムがあり、そのコンソール用に設計されたゲームを実行するためには、システムを**再起動**する必要があります。

では、カーネルは何のために存在するのでしょうか。コンピュータ環境では、プログラムを実行するたびに再起動するのは現実的ではありません。そうすると、各プログラム自体にブートローダやハードウェアの直接制御が必要になります。結局、起動時にプログラムを実行しなければならないのであれば、**OS**などというもの存在しないことになります。

必要なのは、複数のプログラムを実行し、そのメモリ割り当てを管理する機能を提供する抽象化レイヤーです。また、ハードウェアを抽象化することもできます。これは、各プログラムが**OS**なしで起動しなければならない場合には実現できません。結局、ソフトウェアは生のハードウェア上で実行されることになるのです。

ここでのキーワードは「**抽象化**」です。もっと詳しく見てみると....

カーネルの必要性

カーネルは、ハードウェア自体に対する主要な抽象化層を提供します。**カーネルがRing 0であるのは、あらゆるものを直接制御できる**という理由からです。私たちはまだ**Ring 0**にいるので、このことはすでに経験済みです。

これは良いことですが、他のソフトウェアについてはどうでしょうか？私たちが開発しているのは**オペレーティング・エンバイロメント**です。私たちの第一の目標は、アプリケーションやその他のソフトウェアが実行できる、安全で効果的な環境を提供することです。もし、すべてのソフトウェアをカーネルと一緒に**リング0**で動作させることができれば、カーネルは必要ないのではないのでしょうか。そうすると、**リング0のソフトウェアがリング0のカーネルと衝突**して、予想外の結果になる**かもしれません**。結局のところ、彼らはシステムのすべてのバイトを完全にコントロールしているのです。どんなソフトウェアでも、カーネルをはじめ、他のソフトウェアを何の問題もなく上書きすることができる。痛い。

しかし、それは問題の始まりに過ぎません。プログラムやプロセスを切り替えるための共通の土台がないため、マルチタスク、つまり多重処理ができないのだ。一度に実行できるプログラムは1つだけ。

基本的な考え方は、「カーネルは必要不可欠」ということです。他のソフトウェアがすべてを直接コントロールできないようにするだけでなく、カーネルにはそのための**抽象化レイヤー**です。

カーネルがシステムの他の部分とどこで、どのように調和しているかを理解することは非常に重要です。

ソフトウェアの抽象化レイヤー

ソフトウェアには多くの抽象化があります。これらの抽象化は、実装の詳細を隠すだけでなく、コアと基本的なインターフェイスを提供するために行われます。あなたはそれから**守られています**。すべてを直接コントロールできるというのは、一見クールに見えるかもしれませんが、そうすることでどれだけの問題が発生するかを想像してみてください。

どのような問題を指しているのか気になりますよね。電子機器は、基本的に人間が指示したことしかないということを覚えておいてください。私たちは、ソフトウェアから**ハードウェア**、場合によっては**電子機器**に至るまでコントロールすることができます。これらのレベルでミスをすると、物理的にそのデバイスにダメージを与えることになります。

この意味を理解するために、それぞれの抽象化層を見てみましょう。そして、私たちのカーネルがどこに当てはまるのかを見てみましょう。

PModeの保護リングレベルとの関係

ブートローダ3のチュートリアルでは、アセンブリ言語のリングについて詳しく説明しました。また、**プロテクトモード**との関連性についても見てきました。

Ring 0のソフトウェアは、**保護レベルが最も低い**ことを覚えておいてください。つまり、すべてを直接コントロールし、絶対にクラッシュしないことが**期待**されているのです。もし、Ring 0プログラムがクラッシュしたら、システムも一緒にダウンしてしまいます（トリプルフォールト）。

このため、他のすべてのものを直接制御できないようにするだけでなく、ソフトウェアを実行するのに必要な保護レベルのみを与えたいと考えています。このため、通常は

カーネルはリング0で動作（「スーパーバイザーモード

ズ

- デバイスドライバーは、ハードウェアデバイスに直接アクセスする必要があるため、リング1と2
- で動作します。通常のアプリケーションソフトウェアはリング3（「ユーザーモード」）で動作します。

さて...これは一体どういうことなのでしょう？もう少し詳しく見てみましょう...

レベル1：ハードウェアレベル

これが実際の物理的なコンポーネントです。マザーボード上の実際のマイクロコントローラー・チップです。マイクロコントローラーは、他の機器に搭載されているマイクロコントローラーに低レベルのコマンドを送信し、この機器を物理的に制御します。どうやって？それはレベル2で見てみましょう。

ハードウェアの例としては、マイクロコントローラーのチップセット（「マザーボード・チップセット」）、ディスクドライブ、SATA、IDE、ハードドライブ、メモリ、プロセッサ（コントローラーでもある）などが挙げられます（詳細はレベル2を参照）。

これは最も低いレベルであり、純粋なエレクトロニクスであるため最も詳細である。

レベル2：ファームウェアレベル

ファームウェアは、電子機器レベルの上位に位置します。各ハードウェアデバイスやマイクロコントローラーに必要なソフトウェアが含まれています。ファームウェアの例としては、BIOSのPOSTがある。

プロセッサ自体はコントローラに過ぎず、他のコントローラと同様にファームウェアに依存していることを忘れないでください。プロセッサ内の**命令デコーダ**は、1つの機械命令を**マイクロコード**または直接**マイクロコード**に分解します。

詳しくは、「チュートリアル7：システム・アーキテクチャ・チュートリアル」をご覧ください。

マイクロコード

ファームウェアは通常、マイクロコードを用いて開発され、（マイクロアセンブラを用いて）組み立てられて記憶領域にアップロードされるか（BIOSのPOSTなど）、さまざまな手段でデバイスの論理回路にハードワイヤリングされます。

マイクロコードは通常、EEPROMなどのROMチップに格納されています。

マイクロコードは非常にハードウェアに特化しています。新しい変更や改訂があるたびに、新しいマイクロコードの命令セットとマイクロアセンブラを開発する必要があります。マイクロコードは、回路内の個々の電子ゲートやスイッチを制御するために使用されているシステムもあります。そう、それだけローレベルなのです。

マイクロコード

マイクロコードは**非常に**レベルが低く、特にマイクロプロセッサやCPUなどの複雑なシステムでは、開発が**非常に**困難になります。また、コードだけでなく、マイクロプログラムも含めて、何か変更があった場合には、再構築しなければなりません。

このため、いくつかのシステムでは、マイクロコードの上に**マイクロコード**と呼ばれる、より高いレベルの言語を実装しています。この抽象化層のおかげで、MacrocodeはMicrocodeに比べて変更頻度が低く、よりポータブルなものとなっています。また、その抽象化層のおかげで、より簡単に扱うことができます。

しかし、これは非常に低いレベルのものです。これは、上位の機械語をマイクロコードに変換するための内部論理命令セットとして使用され、インストラクションデコーダによって変換される。

レベル3：リング0 - カーネルレベル

これが現在の状況です。ステージ2のブートローダは、カーネルが動作する環境を整えることだけに専念していました。カーネルは、デバイスドライバやアプリケーションソフトウェアと、ハードウェアが使用するファームウェアとの間の抽象化を行います。

レベル4：Ring1および2 - デバイスドライバ

デバイスドライバは、カーネルを経由してハードウェアにアクセスします。デバイスドライバは、特定のマイクロコントローラーを直接制御する必要があるため、かなりの自由度と制御性が求められます。しかし、コントロールが**強すぎると**、システムがクラッシュしてしまいます。例えば、ドライバがGDTを変更したり、独自のGDTを設定したりするとどうなるのでしょうか。そうすると、すぐにカーネルがクラッシュしてしまいます。そのため、これらのドライバが**LGDT**を使って独自のGDTをロードできないようにしたいと思います。これが、これらのドライバが**リング0**ではなく、**リング1**または**リング2**で動作するようにする理由です。

例えば、**キーボードデバイスドライバ**は、**アプリケーションソフトウェア**と**キーボードマイクロコントローラ**の間のインターフェースを提供する必要があります。ドライバは、コントローラに間接的にアクセスするためのルーチンを提供するライブラリとして、カーネルに読み込まれます。

標準的なインターフェイスが使用されている限り、ハードウェアへの依存性をすべて隠すことができるので、非常にポータブルなカーネルを提供することができます。

Level 5: Ring 3 - アプリケーションレベル

ここからがソフトウェアの出番です。システムAPIやデバイスドライバのインターフェースを利用しています。通常、カーネルに直接アクセスすることはありません。

結論

本シリーズでは、カーネルの開発と並行してドライバの開発を行います。これにより、オブジェクト指向を維持しつつ、カーネルの抽象化層を提供することができます。

この点を考慮して、私たちがどのレベルにいるかを考えてみましょう。他のプログラムはすべてカーネルに依存しています。なぜか？カーネルを見てみましょう...

カーネル

カーネルはコアコンポーネントであるため、カーネルに依存するすべてのものの管理を行う必要があります。**カーネルの主な目的は、システムリソースを管理し、他のプログラムがこれらのリソースにアクセスできるようにインターフェースを提供することです。多くの場合、カーネル自身が他のリソースに提供するインターフェースを使用することができません。カーネルは、プログラミングの中で最も複雑で困難な作業であると言われています。**

つまり、良いカーネルを設計・実装するのは非常に難しいということです。

チュートリアル2では、過去の様々なオペレーティングシステムについて簡単に説明しました。このチュートリアルでは、新しい用語の多くを太字にし、チュートリアルの最後にそれらの用語のリストをまとめました。ここからは、そのリストが実際に使われ始めます。

まず、このリストをもう一度見て、カーネルとの関係を見てみましょう。**太字**のものはすべてカーネルが担当しています。

- **メモリ管理** **プログラム管理**
- **マルチタスク**
- **メモリ保護**

2021/11/15 13:07

- **Fixed Base Address** - チュートリアル2で説明しました。
- **Multiuser** - これは通常、シェ尔によって実装されます。

オペレーティングシステム開発シリーズ

- カーネル - もちろん
- 、ファイルシステム
- コマンドシェル
- グラフィカル・ユーザー・インタ
- ーフェース (GUI) グラフィカル・シェル
- リニア・ブロック・アドレッシング (LBA) - チュートリアル2で説
- 明しました ブートローダ -完成

上記のうちのいくつかは、カーネルが使用する別個のドライバとして実装することができます。例えば、WindowsではNTFSファイルシステムドライバとして**ntfs.sys**を使用しています。

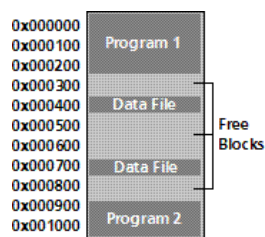
このリストは、**チュートリアル2**で見覚えがあるはずです。また、これらの用語のいくつかを取り上げました。**太字**の用語を見て、それらがどのようにカーネルに関連しているかを見てみましょう。また、いくつかの新しい概念についても見ていきます。

メモリ管理

これはカーネルの中でも最も重要な部分です。すべてのプログラムとデータにはこれが必要です。ご存知のように、カーネルは**スーパーバイザーモード**（リング0）のままなので、**メモリ内のすべてのバイトに直接アクセスできます**。これは非常に強力ですが、問題も生じます。特に、複数のプログラムやデータがメモリを必要とするマルチタスク環境では、問題が生じます。

私たちが解決しなければならない主要な問題の1つはメモリが足りなくなったらどうしよう？

もうひとつの問題は**フラグメンテーション**です。**ファイルやプログラムをメモリの連続した領域に読み込むことができない場合があります**。例えば、2つのプログラムがロードされているとします。1つは0x0で、もう1つは0x900です。これらのプログラムは両方ともファイルのロードを要求していたので、データファイルをロードします。



ここで何が起きているのかを見てみましょう。これらすべてのプログラムとファイルの間には、未使用のメモリがたくさんあります。では、さらに大きなファイルを追加して、上記に収まらなくなったらどうなるでしょう？ここで、現在の方式では大きな問題が発生します。現在実行中のプログラムや読み込まれているファイルを破壊してしまうため、特定の方法で直接メモリを操作することはできません。

また、各プログラムがどこでロードされるかという問題もあります。各プログラムは**Position Independent**であるか、または**リロケーションテーブル**を提供する必要があります。これがないと、プログラムがどのベースアドレスにロードされるのかわからない。

もっと詳しく見てみましょう。**ORG**指令を覚えていますか？この指令は、プログラムがどこからロードされるかを設定します。違う場所でプログラムをロードすると、プログラムは間違ったアドレスを参照してしまい、クラッシュしてしまいます。この理論を簡単に試すことができます。今、**Stage2**は0x500でロードされることを期待しています。しかし、**Stage1**の中で0x400にロードすると（**Stage2**の中で**ORG0x500**を維持したまま）、トリプルフォールトが発生してしまいます。

これには2つの新しい問題が加わります。プログラムをどこにロードするか、どうやって知ることができるのか？私たちが持っているものがすべてバイナリイメージであることを考えると、**知ることはできません**。しかし、すべてのプログラムが同じアドレス（たとえば0x0）から始まることを標準とすれば、知ることができます。これはうまくいきますが、マルチタスクをサポートしようとすると、実行するのは不可能です。しかし、**それぞれのプログラムに、事実上0x0から始まる独自のメモリ空間を与えれば、これはうまくいきます**。

結局のところ、各プログラムから見れば、実際の（物理的な）メモリでは違っていても、すべて同じベースアドレスでロードされています。そこで必要になるの

が、物理メモリを抽象化する方法です。もう少し詳しく見てみましょう。

仮想アドレス空間 (VAS)

仮想アドレス空間とは、**プログラムのアドレス空間**のことである。ここで注意しなければならないのは、これは**システムメモリとは関係がない**ということです。これは、**各プログラムがそれぞれ独立したアドレス空間を持つようにするためのものです**。これにより、あるプログラムが別のプログラムにアクセスできないようにしている。

VASは仮想的なものであり、物理的なメモリを直接使用するものではないため、ディスクドライブなどの他のソースをメモリのように使用することができます。つまり、**システムに物理的に搭載されている以上のメモリを使用することができるのです**。

これにより、「メモリが足りない」という問題が修正されます。

また、各プログラムが独自の**VAS**を使用しているため、各プログラムは常にベース0x0000:0000から始まるようにすることができます。これにより、前述の再配置の問題や、メモリの断片化が解消され、各プログラムに連続した物理ブロックを割り当てる心配がなくなります。

仮想アドレスは、MMUを介してカーネルによってマッピングされます。これについては、もう少し後に説明します。

仮想メモリ。概要

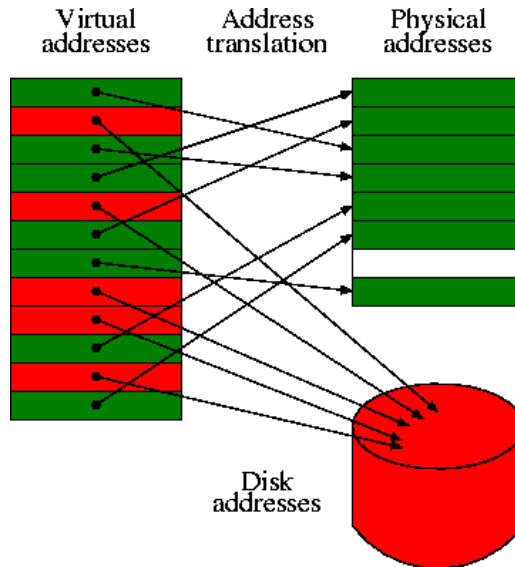
仮想メモリーは、ハードウェアとソフトウェアの両方で採用されている特別な**メモリーアドレス方式**です。これにより、連続していないメモリを連続したメモリのように扱うことができます。

Virtual Memoryは、**Virtual Address Space**の概念に基づいています。すべてのプログラムに独自の仮想アドレス空間を提供することで、メモリの保護やメモリの断片化を防ぐことができます。

仮想メモリは、システムに実際に搭載されているよりも多くのメモリを間接的に使用方法でもあります。一般的な方法としては、**ハードドライブ**に保存されている**ページファイル**を使用する方法があります。

仮想メモリは、ハードウェアレベルで処理されるため、動作させるためにはハードウェアデバイスコントローラを介してマッピングする必要があります。これは通常、後述する**MMU**を通じて行われます。

仮想メモリの使用例として、実際に見てみましょう。



ここで何が起きているのかに注目してください。**仮想アドレス**内の各メモリブロックは直線的ですが、各メモリブロックは、実際の物理的な**RAM**内の位置か、ハードディスクなどの別のデバイスに**マッピング**されています。このブロックは、必要に応じてこれらのデバイス間でスワップされます。これは遅く見えるかもしれませんが、MMUのおかげで非常に高速です。

覚えておいてください。各プログラムは、上図のような**仮想アドレス空間**を持ちます。各アドレス空間は0x0000:00000から始まる直線的なものであるため、メモリの断片化やプログラムの再配置に関する問題のほとんどが解決されます。

また、**仮想メモリ**はメモリブロックを使用するデバイスが異なるため、システム内のメモリ量以上の管理を容易に行うことができます。例えば、システムメモリが足りなくなった場合、代わりにハードディスクにブロックを割り当てることができます。メモリが足りなくなったら、必要に応じてこのページファイルを増やすか、警告やエラーのメッセージを表示することができます。

各メモリの「ブロック」は「ページ」と呼ばれ、通常**4096バイト**の大きさがあります。

繰り返しになりますが、すべての内容は後ほど詳しく説明します。

メモリマネジメントユニット (MMU) 。概要

おやおや、どこかで聞いたことのあるような言葉ですねo.o :)

MMUは、**Paged Memory Management Unit (PMMU)**とも呼ばれ、マイクロプロセッサ内のコンポーネントで、CPUから要求されたメモリの管理を行います。**仮想アドレスから物理アドレスへの変換、メモリ保護、キャッシュ制御など**、さまざまな役割を担っています。

セグメンテーション。概要

Segmentationは**Memory Protection**の手法の一つです。Segmentationでは、現在実行中のプログラムから特定のアドレス空間のみを割り当てます。これは、**ハードウェアレジスタ**を通じて行われます。

セグメント化は、最も広く使われているメモリ保護方式の一つです。x86では、通常、**セグメントレジスタ**によって処理されます。CS、SS、DS、ES。

Real Modeを通してその使い方を見てきました。

ページング。概要

THIS is important to us.ページングとは、RAM上にはない仮想メモリページへのプログラムのアクセスを管理するプロセスです。これについては後ほど詳しく説明します。

プログラママネジメント

ここからはリングのレベルが重要になってきます。

ご存知のように、当社のカーネルはRing 0で、アプリケーションはRing 3である。これは、アプリケーションが特定のシステムリソースに直接アクセスするのを防ぐという意味では良いことです。しかし、これらのリソースの多くはアプリケーションが必要とするものなので、これは悪いことでもあります。

プロセッサがどのようにして自分のリングレベルを知り、どのようにしてリングレベルを切り替えるのか、興味があるかもしれません。プロセッサは単純に内部フラグを使って現在のリングレベルを保存しています。では、プロセッサはどのリングでコードを実行するかをどうやって知っているのでしょうか？

ここで重要になるのが、**GDTとLDT**です。

ご存知の通り、リアルモードでは保護レベルがありません。そのため、すべてが「Ring 0」となります。**プロテクトモードに入る前に、GDTを設定しなければならないことを覚えておいてください。**また、32ビットモードに入るためには、**ファージャンプ**を実行する必要があることも覚えておいてください。これらは非常に重要な役割を果たしますので、ここで詳しく説明しましょう。

スーパーバイザーモード

リング0は**スーパーバイザーモード**として知られています。このモードでは、すべての命令、レジスタ、テーブルなど、より高いリングレベルのアプリケーションがアクセスできない特権的なリソースにアクセスできます。

リング0は、**カーネルレベル**とも呼ばれ、絶対に故障しないことが**期待**されています。リング0のプログラムがクラッシュすると、システムも一緒にダウンしてしまいます。それを覚えておいてください。

「**大きな力には大きな責任が伴う**」。これがプロテクトモードの一番の理由です。」

スーパーバイザーモードでは、システムレベルのソフトウェアで変更可能なハードウェアフラグを利用します。システムレベルのソフトウェア（リング0）にはこのフラグが設定されますが、アプリケーションレベルのソフトウェア（リング3）には設定されません。

Ring0コードだけができて、Ring3コードができないことがたくさんあります。^ズチュートリアル7のフラグレジスタを覚えていますか？RFLAGSレジスタの**IOPL**フラグは、**IN**命令や**OUT**命令などの特定の命令を実行するために必要なレベルを決定します。IOPLは通常0なので、**リング0プログラムのみがソフトウェアポートを介してハードウェアに直接アクセスできる**ことになります。このため、頻繁にRing 0に切り替える必要があります。

カーネルスペース

カーネルスペースとは、カーネルとRng 0デバイスドライバのために確保されたメモリの特別な領域を指します。ほとんどの場合、**カーネルスペースは、仮想メモリのようにディスクにスワップアウトしてはいけません**。

OSがユーザースペースで動作する場合、「ユーザーランド」と呼ばれることがあります。

ユーザースペース

これは通常、**リング3のアプリケーションプログラム**です。各アプリケーションは、通常、独自の**仮想アドレス空間（VAS）**で実行され、異なるディスクデバイスからスワップすることができます。**各アプリケーションはそれぞれの仮想メモリ内にあるため、他のプログラムのメモリに直接アクセスすることはできません**。このため、Ring 0プログラムを経由してアクセスする必要があります。これは**デバッガ**には必須のことです。

アプリケーションは通常、最も権限のないものです。そのため、システムリソースにアクセスするには、通常、リング0のカーネルレベルのソフトウェアにサポートを求める必要があります。

保護レベルの切り替え

必要なのは、これらのアプリケーションがシステムにこれらのリソースを照会できるようにすることです。しかし、そのためには、Ring 3ではなく**Ring 0**である必要があります。このため、プロセッサの状態をRing 3からRing 0に切り替えて、アプリケーションがシステムに問い合わせできるようにする方法が必要です。

チュートリアル5でアセンブリ言語のリングについて説明しました。以下の条件で、プロセッサが現在のリング・レベルを変更することを覚えておいてください。

- ・ **ファージャンプ**、**ファークール**、**ファットレイト**などの指示命令。
- ・ **INT**, **SYSCALL**, **SYSEXIT**, **SYSENTER**, **SYSRETURN** などのトラップ命令。
- ・ **例外**

つまり、アプリケーションが（リング0に切り替えながら）システムルーチンを実行するためには、**ファージャンプ**するか、**インタラプト**を実行するか、**SYSENTER**などの特別な命令を使用する必要があります。

これは素晴らしいことですが、プロセッサはどのリングレベルに切り替わるかをどうやって知るのでしょうか？ここで、**GDT**の出番となります。

GDTの各ディスクリプターに、**リングレベル**を設定する必要があったことを覚えていますか？現在のGDTでは、2つの記述子があります。それぞれがカーネルモードのリング0で、**これがカーネルスペースです**。

必要なのは、現在のGDTに2つのモード記述子を追加することだけですが、**リング3のアクセス用に設定されています**。これが**私たち**

のユーザースペースです。もう少し詳しく見てみましょう。

チュートリアル8で、ここで**重要なのはアクセスバイトであることを思い出してください**。このため、再びバイトパターンを示します。

- ・ ビット0（GDTではビット40）。アクセスビット（仮想メモリで使用）。仮想メモリを使用していないので（まだ、とにかく）、無視します。したがって、この
- ・ ビットは0です。第1ビット（GDTの第41ビット）：読み取り/書き込み可能ビットです。コードセクタのために）セットされているので、セグメント（0x0から0xFFFFまで）のデータをコードとして読み込んで実行することができます。
- ・ 第2ビット（GDTでは第42ビット）：「拡張方向」のビットです。これについては後ほど詳しく見ていきます。今のところは無視してください。
- ・ 第3ビット（GDTでは第43ビット）：これがコードまたはデータの記述子であることをプロセッサに伝える。（セットされているので、コード記述子であることを示しています。）
- ・ ビット4（GDTではビット44）。システムまたは「コード/データ」の記述子として表します。これはコードセクターなので、この
- ・ ビットは1に設定されています。第5-6ビット（GDTの第45-46ビット）：特権レベル（Ring 0またはRing 3など）です。今回はリング0なので、両ビットとも0です。
- ・ ビット7（GDTではビット47）。セグメントがメモリ内にあることを示すために使用されます（仮想メモリで使用）。まだ仮想メモリを使用していないので、今は0に設定する

```

;*****
グローバルディスクリプターテーブル (GDT)
;*****

gdt_data:

; Nullディスクリプター (Offset: 0x0)--Remember each descriptor is 8
    bytes! dd                                0; null descriptor
    dd 0

カーネルスペースコード (オフセット: 0x8バイト
    dw                                0FFFFh; リミット・ロー
    dw                                0; ベースロー
    db                                0; ベース・ミドル
    db                                10011010b; access - ビット5と6 (特権レベル) がリング0の場合は0であることに注意
    db                                11001111b; granularity
    db                                0; base high

; カーネルスペースデータ (オフセット: 16 (0x10) バイト
    dw                                0FFFFh; limit low (Same as code)10:56 AM 7/8/2007
    dw                                0; ベースロー
    db                                0; ベース・ミドル
    db                                10010010b; access - Ring 0ではビット5と6 (特権レベル) が0であることに注意 db
    db                                11001111b; granularity
    db                                0; base high

ユーザースペースコード (オフセット: 24 (0x18) バイト
    dw                                0FFFFh; リミット・ロー
    dw                                0; ベースロー
    db                                0; ベース・ミドル
    db                                11111010b; access - ビット5と6 (特権レベル) がRing 3では11bであることに注意 db
    db                                11001111b; granularity
    db                                0; base high

; ユーザースペースデータ (オフセット: 32 (0x20) バイト
    dw                                0FFFFh; limit low (Same as code)10:56 AM 7/8/2007
    dw                                0; ベースロー
    db                                0; ベース・ミドル
    db                                11110010b; access - ビット5と6 (特権レベル) がRing 3では11bであることに注意してくださ
    い。

```

ここで何が起こっているのかに注目してください。すべてのコードとデータは同じ範囲の値を持っています - 唯一の違いは**リングレベル**の違いです。

ご存知のように、**プロテクトモード**ではCSに**CPL (Current Privilege Level)**を格納します。プロテクトモードに初めて入るとき、**リング0**に切り替える必要がありました。CSの値が無効(リアルモードから)だったため、GDTからCSに正しいディスクリプタを選択する必要があります。詳細は**チュートリアル8**を参照してください。

これには、新しい値をCSにアップロードする必要があるため、**ファージャンプ**が必要でした。Ring 3の記述子に**ファージャンプ**することで、実質的にRing 3の状態に入ることができます。

ご存知のように、**INT、SYSCALL/SYSEXIT/SYSENTER/SYSRET、ファークール、または例外**を使って、プロセッサをリング0に戻すことができます。

これらの方法について詳しく見てみましょう。

システムAPI。アブストラクト

プログラムは、システムリソースにアクセスするために**System API**に依存しています。ほとんどのアプリケーションは、システムAPIを直接参照するか、**Cランタイム・ライブラリ**などの言語APIを介して参照します。

システムAPIは、**システムコール**を通じてアプリケーションとシステムリソースの間の**インターフェース**を提供します。

割り込み

ソフトウェアインタラプトとは、ソフトウェアに組み込まれた特別なタイプの割り込みです。割り込みは非常に頻繁に使用され、特別なテーブルを使用しています。**割り込み記述子テーブル (IDT)**。割り込みについては、後ほど詳しく説明しますが、これはカーネルに実装する最初のものです。Linuxでは、すべてのシステムコールにINT 0x80を使用しています。

割り込みは、システムコールを実装するための最もポータブルな方法です。 このため、システムルーチンを呼び出す最初の方法として、割り込みを使用することになります。

コールゲート

コールゲートは、Ring 3アプリケーションがより優先度の高い(Ring 0,1,2)コードを実行する方法を提供します。**コールゲート**はRing 0ルーチンとRing 3アプリケーションの間のインターフェースであり、通常はカーネルによって設定されます。

コールゲートは、ファールコールへの単一のゲート(エントリーポイント)を提供します。このエントリーポイントは、GDT または LDT 内で定義されます。コールゲートを理解するには、例を挙げて説明する方がはるかに簡単です。

```
;*****
;グローバルディスクリプターテーブル (GDT)
;*****

gdt_data:

; Nullディスクリプター (Offset: 0x0) --Remember each descriptor is 8
bytes! dd 0; null descriptor
dd 0

カーネルスペースコード (オフセット: 0x8バイト)
dw 0FFFFh; リミット・ロー
dw 0; ベースロー
db 0; ベース・ミドル
db 10011010b; access - ビット5と6 (特権レベル) がリング0の場合は0であることに注意
db 11001111b; granularity
db 0; base high

; カーネルスペースデータ (オフセット: 16 (0x10) バイト)
dw 0FFFFh; limit low (Same as code)10:56 AM 7/8/2007
dw 0; ベースロー
db 0; ベース・ミドル
db 100110010b; access - Ring 0ではビット5と6 (特権レベル) が0であることに注意 db
db 11001111b; granularity
db 0; base high

; Gate1 (Offset: 24 (0x18) bytes)
dw (Gate1 & 0xFFFF) dw 0x8 ゲートルーチンのリミットロードアドレス
db 0 コード・セグメント・セクター
db 11101100b ベース、ミドル
db 0 アクセス - Ring 3では、ビット5と6 (特権レベル) が11であることに注意してください。
db (Gate1 >> 16) グラニューラリティ
ゲートルーチンのベースハイ

GDTの終わり。どこでもルーチンを定義する

;コールゲートルーチンGate1

です。

Ring 3では特別なことをしています。

retf 呼び出したルーチンに戻る
```

上記はコールゲートの一例です。

コールゲートを実行するには、GDT内の**ディスクリプターコード**からオフセットします。これが、**jmp 0x8:Stage2**命令といかに似ているかに注目してください。

コールゲートの実行

call 5x0x18:0

ファークール...私たちのGate1ルーチンを呼び出す

コールゲートは、最近のOSではあまり使われていません。その理由の一つは、ほとんどのアーキテクチャがコールゲートをサポートしていないからです。また、コールゲートは**FAR CALL**命令と**FAR RET**命令を必要とするため、非常に低速です。

GDTが保護されたメモリにないシステムでは、他のプログラムが独自のコールゲートを作成して保護レベルを上げる（リング0アクセスを得る）ことも可能です。これらのコールゲートにはセキュリティ上の問題があることも知られています。例えば、注目すべきワームの1つである**Gurong**は、Windows OSに独自のコールゲートをインストールします。

SYSENTER / SYSEXIT 命令

これらの命令は、Pentium II以降のCPUから導入されました。最近のAMDプロセッサでもこれらの命令をサポートしているものがあります。

SYSENTERはどのアプリケーションでも実行できます。**SYSRET**はRing 0のプログラムでのみ実行可能です。

これらの命令は、ユーザーモード（Ring 3）から特権モード（Ring 0）に制御を素早く移行させるための高速な方法として使用されます。これにより、ユーザーモードからシステムルーチンを高速かつ安全に実行することができます。

これらの命令は**MSR（Model Specific Registers）**に直接依存しています。**MSR**と**RDMSR**および**WRMSR**命令の説明については、チュートリアル7を参照してください。

SYSENTER

CS = IA32_SYSENTER_CS MSR + 値 8

- esp = ia32_sysenter_esp msr eip
- ss = ia32_sysenter_ip msr ss =
- ia32_sysenter_ss msr

この命令は、Ring 3コードからRing 0に制御を移すためにのみ使用されます。起動時には、これらのMSRを**スターティング・ロケーション**に設定する必要があります。は、すべてのシステムコールの**エントリーポイント**となります。

それでは、SYSEXITについて見てみましょう。

SYSEXIT

CS = IA32_SYSENTER_CS MSR + 値16 **SYSEXIT**命令は、以下のレジスタをMSRで定義された位置に自動的に設定

- します。
- ESP = ECXレジスタ
- EIP = EDXレジスタ
- ss = ia32_sysenter_cs msr msr + 24

SYSENTER/SYSEXITの使用

さて、この手順を使うと複雑に見えるかもしれませんが、それほど難しくはありません ;)

SYSENTERとSYSEXITは、それらを呼び出す**前に**MSRが設定されている必要があるため、まずそれらのMSRを初期化する必要があります。

MSR内のIA32_SYSENTER_CSがインデックス0x174、IA32_SYSENTER_ESPが0x175、IA32_SYSENTER_IPが0x176であることを覚えておいてください。チュートリアル7を思い出してください。

これを踏まえて、SYSENTERに設定してみましょう。

```
%define IA32_SYSENTER_CS 0x174
%define IA32_SYSENTER_ESP
0x175
%define IA32_SYSENTER_EIP
0x176    eax, 0x8                                カーネルコード記述子
mov     edx, 0
mov     ecx,
wrmsr   IA32_SYSENTER_CS

        MOVEAX, ESC
        movedx, 0

movecx, IA32_SYSENTER_ESP
wrmsr

moveax, Sysenter_Entry
        movedx, 0
movecx, IA32_SYSENTER_EIP
wrmsr

; ここで、リング0のプログラムまたはリング3のプログラムからリング0でSysenter_Entryを実行するためにsysenterを使用することができます: sysenter

Sysenter_Entryです。

シスコンがここでジャンプすると、プリビレッジレベル0でこのコードを実行することになる。コールゲートと同様に、通常、私たちはは、すべてのシステムコールに対して単一のエントリーポイントを提供します。
```

sysenterを実行するコードがRing 3で、**Sysenter_Entry**が保護レベル0の場合、プロセッサは**SYSENTER**内でモードを切り替えます。の指示を受けています。

上のコードでは、両方とも保護レベル0になっているので、プロセッサはモードを変更することなくルーチンを呼び出します。ご

覧のように、SYSENTER.\やSYSEXITを呼び出す前に、ちょっとした作業が必要になります。

SYSENTERとSYSEEXITはポータブルではありません。このため、SYSENTER/SYSEEXITと一緒に、よりポータブルな別の方法を導入するのが賢明です。

SYSCALL / SYSRET 命令

[近々、ここにSYSCALLとSYSRETのセクションを追加する予定です]。

エラー処理

あるプログラムが問題を起こした場合、どうすればいいのか？その問題が何なのか、どう対処すればいいのかをどうやって知のでしょうか？

通常、これは**例外処理**によって行われます。0で割るなどの無効な命令によってプロセッサが無効な状態になると、プロセッサは**割り込みサービスルーチン (ISR)**を起動します。独自のISRをマッピングしていれば、私たちのルーチンが呼び出されます。

呼び出される**ISR**は、問題が何であったかによって異なります。これは素晴らしいことです。問題が何であるかがわかっているのに、元々問題を起こしたプログラムを探して見ることができます。

その方法の一つは、プロセッサ時間を与えた最後のプログラムを取得することです。これは、ISRを生成したプログラムであることが保証されています。

プログラムの情報が得られれば、エラーを出力したり、プログラムの停止を試みたりすることができます。

IRQは、プロセッサ内部の**PIC (Programmable Interrupt Controller)** によってマッピングされます。**IRQ**はプロセッサ内部の**PIC (Programmable Interrupt Controller)** によってマッピングされ、**IDT (Interrupt Descriptor Table)** 内の割り込みエントリにマッピングされます。これはカーネルの中で最初に扱うものなので、後ですべてを説明します。

結論

このチュートリアルでは、カーネル理論、メモリ管理の概念、**VMA (Virtual Memory Addressing)**、プログラム管理、Ring 0とRing 3の分離、アプリケーションとシステムソフトウェアの間のインターフェースの提供など、さまざまな概念を見てきました。ふう。これはすごいことだと思いませんか？

このチュートリアルに出てくる概念の多くは初めて聞くものかもしれませんが、心配しないでください。このチュートリアルは「**Get your feet wet**」と題して、カーネルに関する基本的な概念をすべてカバーしています。

このチュートリアルでは、カーネルがしなければならないことの表面をわずかにぞっただけです。とはいえ、これがスタートです。)

次のチュートリアルでは、カーネルを別の視点から見ていきます。カーネルの設計と実装について、新しい概念を説明します。その後、CやC++を扱うためのコンパイラやツールチェーンの構築を開始します。面白そうでしょ？

現在、私のカーネルには**MSVC++ 2005**を使用しています。

また、**マルチタスク**、**TSS**、**ファイルシステム**など、ここでは見ていなかったコンセプトも仕上げていきます。楽しくなりそうですね。)

次の機会まで。

マイク
BrokenThorn Entertainment社。現在、DoEとNeptuneOperating Systemを開発中です。質問

やコメントはありますか？お気軽にお問い合わせください。

あなたも記事の改善に貢献したいと思いませんか？もしそうなら、ぜひ私に教えてください。



第11章

ホーム



第13章



オペレーティングシステム開発シリーズ

オペレーティングシステム開発 - カーネル。基礎知識編2

by Mike, 2007

このシリーズは、オペレーティングシステムの開発を一から実演し、教えることを目的としています。

はじめに

歓迎します。:)

前回のチュートリアルでは、多くの概念を説明しました。このチュートリアルでは、引き続きこれらのコンセプトを見ていきます。このチュートリアルでは、非常に重要なコンセプトをいくつか取り上げます。今日のメニューは以下の通りです。

- ハードウェア・アブストラク
- ション・カーネル。新しい視点
- カーネルデザイン概要：一次設計モデル カーネルの設計
- 概要：二次設計モデル

このチュートリアルでは、ブートローダと初歩的なカーネル設計の間の接着剤について説明します。このチュートリアルは、カーネルとは何か、何をしなければならないかを理解するために、すべてをまとめるための接着剤となります。

ここで紹介したコンセプトは、次の数回のチュートリアルで非常に重要になります。ハードウェア抽象化レイヤー（HAL）と超1337カーネルの設計と開発を開始するからです。

それでは、ゆったりとしたシートに座って、今回も楽しいハッピーなチュートリアルを見てみましょう。そうそう、このチュートリアルもそんなに大きくないのがいいですよね :)

注：このチュートリアルは、読者が先に[チュートリアル2](#)を読んでいることを推奨しています。

私はexokernelsを説明するセクションを追加する予定です。これは全く新しいカーネルデザインのコンセプトです。このシリーズの中でこのデザインを実装するつもりはありませんが、読者の皆さんがこのカーネルデザインに興味を持ってくれるのではないかと考えています。もしかしたら、このデザインを自分のOSに使うかもしれません。

いいですか？

ハードウェアの抽象化

ハードウェアの抽象化は非常に重要です。ハードウェアのプログラミングがいかに複雑で、いかにハードウェアに依存しているかは、もうお分かりいただけたと思います。そこで登場するのが、**Hardware Abstraction Layer (HAL)**です。

HALとは、物理的なハードウェアへのインターフェースを提供するために使用されるソフトウェアの抽象化層のことです。抽象化された層である。これらの抽象化は、特定のデバイスやコントローラの詳細を知る必要がなく、デバイスと対話する方法を提供します。

通常、最近のOSでは、HALは基本的なマザーボード・チップセット・ドライバです。これは、カーネルと、プロセッサを含むマシンのハードウェアとの間の基本的なインターフェースを提供します。これは素晴らしいことで、カーネルはハードウェアへのアクセスが必要なときはいつでもHALと対話することができます。これはまた、カーネルが完全にハードウェアに依存しないことを意味します。

また、特定のコントローラやマッピングではなく、デバイスそのもので考えることができます。これにより、カーネル自体もクリーンになります。

もうひとつの大きなメリットは、抽象化そのものにあります。OSを異なるハードウェアのシステムに移植する場合、必要なのはそのために新しいHALを開発することだけです。ただし、これはHALの設計が非常に優れていることが前提です。

最近のOSのほとんどは、何らかの形でHALを使用しています。ここでは、チップセットハードウェアとカーネルの間でマザーボードチップセットドライバとして機能するHALを開発します。次のチュートリアルでは、HALの背後にあるプロセッサ自体を抽象化することで、HALの開発を開始します。

カーネル。新しい視点

では、そもそもカーネルとは何か。カーネルは、John N. Shutt氏 (Serously ;)が開発したSchemeライクなプログラミング言語です。

とにかく、別の定義を見てみましょう。カーネルとは、システムの中心となるコンポーネントです。このシステムは何でもよいのです。カーネルはシステムの中核であり、システムの効率的な実行を管理するための非常に基本的な機能を提供します。

オペレーティングシステムでは、この優れたカーネルが、システムのハードウェアやリソースに対する最も基本的なインターフェースを提供する。また、プロセッサ管理、I/O管理、メモリ管理、プロセス管理など、最も基本的な管理機能も提供します。カーネルは、開発するシステムの複雑さに応じて、さらに多くのものを含むことができます。

さて、前のリストには見覚えがあるかもしれませんが、どこかで見たことがあるような？[チュートリアル2](#)の中で実際に見てみました。

もっとよく理解するために、もう少し詳しく見てみましょう。

カーネル。すべてをまとめる

メモリ管理

さて、それでは[チュートリアル2](#)の内容を思い出してください。メモリの管理と保護に関する項目の基本的なリストを作成しました。これをもう一度見てみましょう。

メモリマネジメントとは

- メモリを要求するプログラムに対して、動的にメモリを与えたり、使ったりすること。
- ページング、あるいは仮想メモリのようなものです。
- OSカーネルが未知のメモリや無効なメモリを読み書きしないようにする。
- Memory Fragmentationの監視と処理。

メモリープロテクションとは

- プロテクトモードで無効なディスクリプターにアクセスする（または無効なセグメントアドレスにアクセスする）プログラム自体を上書きする。
- メモリ上の他のファイルの一部または全部を上書きすること。

今頃は、[チュートリアル2](#)のすべての内容を理解しているはずです。

リング0で実行されているカーネルは、メモリ内のすべてのバイトを直接制御していることを忘れてはいけません。また、これらはすべて物理的なメモリ上で直接実行されていることも覚えておいてください。もしメモリが足りなくなったらどうなるでしょうか？存在しないメモリに書き込んだらどうなるでしょうか？ハードウェア・デバイスが使用するメモリ・ロケーションはどうなるのか？また、メモリ内に存在する「隙間」にも触れていません。

警告物理メモリ上のランダムな位置に書き込みを行うと、（書き込み先によっては）誤動作の原因になったり（そのメモリ領域を使用しているハードウェアデバイスによっては）、システムが完全に起動できなくなったり、全く使い物にならなくなったりすることがあります。（また、BIOSのデータ領域に書き込みを行うと、システムが起動しなくなり、全く使えなくなります。

このように考えると、物理的なメモリを適切に管理することがいかに重要であるかがわかんと思います。

物理的なメモリを適切に管理するカーネルは、アプリケーションとメモリの間に仮想的なインターフェースを作り出すことができます。これは、ユーザースペースとカーネルスペースのコードとデータを分離することと、仮想アドレスによって可能になります。

物理的なメモリで直接実行することには多くの問題があるため、より優れた方法でメモリをエミュレートすることができます。このメモリエミュレーション（仮想メモリ）は、物理的なRAMよりも多くのメモリを持つシステムをエミュレートすることができ、各アプリケーションは独自の仮想メモリアドレス空間を使用します。

プロセッサマネージメント

これは新しいものです。ご存知の通り、BIOS ROMはプライマリ・プロセッサを初期化して起動します。シングルコアしか起動しません。ミューティコア・プロセッサを搭載したシステムや、複数のプロセッサを搭載したシステムでOSを動作させる場合は、他のプロセッサやコアを手動で起動させる必要があります。

アプリケーションが異なるプロセッサをいつでも操作できるようにすると、システムに致命的な問題が発生する可能性があります。そのため、アプリケーションにこのような機能を持たせてはいけません。

I/Oデバイスの管理

物理的なメモリと同様に、アプリケーションがコントローラのポートやレジスタに直接アクセスすると、コントローラの誤動作やシステムのクラッシュの原因となります。また、デバイスの複雑さによっては、プログラムが驚くほど複雑になり、複数の異なるコントローラを使用しているものもあります。そのため、デバイスを管理するためのより抽象的なインターフェイスを提供することが重要です。このインターフェイスは、通常、**デバイスドライバ**や**ハードウェア抽象化レイヤー**によって行われます。これにより、デバイスの詳細ではなく、デバイスの観点から考えることができます。

アプリケーションがこれらのデバイスへのアクセスを必要とすることはよくあります。カーネルは、何らかの方法でシステムにデバイスを問い合わせることで、これらのデバイスのリストを維持する必要があります。アプリケーションがデバイスに対する操作（例えば、文字の表示など）を要求すると、カーネルは現在アクティブなビデオドライバにこの要求を送信する必要があります。ビデオドライバは、この要求を実行する必要があります。これは**IPC (Inter Process Communication)** の一例です。

プロセスマネージメント

これはカーネルの、そしてあらゆるコンピュータの、最も重要なタスクです。カーネルには、実行時間を割り当てたり、さまざまなアプリケーションやプロセスを実行・管理する方法が必要です。

そこで登場するのが「プログラム・マネージメント」と「マルチタスキング」です。これらの用語は[チュートリアル2](#)でお馴染みのものです。チュートリアル2の内容をもう一度見てみましょう。

プログラムマネージメントは以下の責任を負います。

- プログラムが他のプログラムを上書きしないようにする。プログ
- ラムがシステムデータを破壊しないことを保証する。
- タスクを完了するためにプログラムからの要求を処理する（メモリの割り当てや解放など）。

マルチタスキングとは

- 複数のプログラムを切り替えて、一定の時間内に実行する。タスクマネージャーを用意
- して切り替えを可能にする（Windowsのタスクマネージャーなど）。TSS (Task State
- Segment) 切り替え。またまた新語です
- 複数のプログラムをシミュレートして実行する。

アプリケーションを実行するには、カーネルがアプリケーションの**仮想アドレス空間 (VAS)**を設定し、ファイルをVASにロードする必要があります。その後、カーネルはアプリケーションのスタックを設定し、そこにジャンプして実行を開始します。

Virtual Addressingによって、アプリケーションがシステムメモリの問題を起こさないようにすることができます。

マルチタスクシステムでは、タスクマネージャは各プロセスに一定の時間を割り当て、その時間内にのみ実行します。そして、実行中のアプリケーションを切り替えていきます。割り当てられる時間が少ないので、タスクマネージャは実行中のプロセスを素早く切り替えることができ、複数のプロセスが同時に実行されているように見せかけることができます。

これは、ハードウェアまたはソフトウェアによって行われます。プロセッサは、**TSS (Task State Segment)** レジスタを使用して、ハードウェアによるタスク切り替えをサポートしています。

システムAPI

ここまでくると、すべてがどのように組み合わされているのか、また、[チュートリアル2](#)のコンセプトがどこから入ってくるのか、理解できるようになっているはずです。しかし、まだカバーしていない小さなディテールがあります。

アプリケーションはどのようにしてカーネルにデバイスやシステムリソースへのリクエストを行うのでしょうか？これまで、OSがアプリケーションを管理・制御する方法を見てきましたが、アプリケーションはどのようにシステムを制御するのでしょうか？

ここで登場するのが、システムAPI (**Application Programming Interface**) です。システムAPIは、アプリケーションがカーネルやその他のシステムソフトウェアと対話するために使用できるAPIです。

システムAPIを作成するには、さまざまな方法があります。ほとんどのシステムでは、システムAPIのルーチンを割り込みでサポートしています。例えば、LinuxカーネルのシステムAPIでは、システムルーチンに主に割り込み番号0x80を使用しています。

結論

いやあ、すごい量ですね。まだ理解できなくても気にしないでください。すぐに理解できるようになりますよ！)

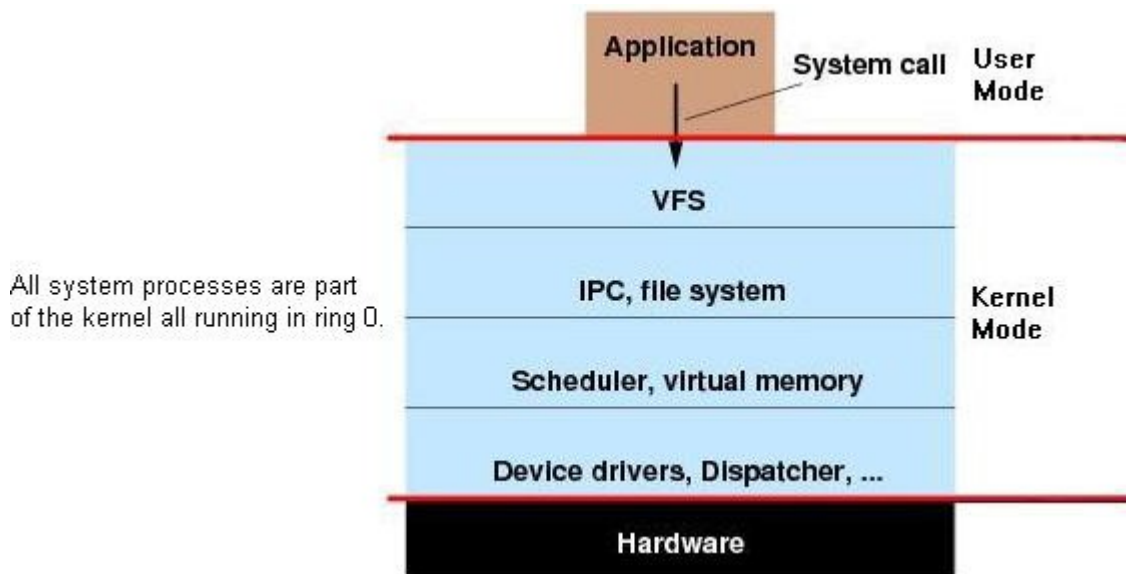
カーネルデザイン - 概要：一次設計モデル

ここまでくると、カーネルがいかに重要で、どこに組み込まれているかがわかってくるのではないのでしょうか。

様々なデザインやセットアップを用いて開発されたオペレーティングシステムは数多くあります。これらのデザインの多くは、基本的なコンセプトが似通っています。

カーネルの構造には様々な方法があります。ここでは、よく使われる設計をいくつか紹介します。

モノリシック・カーネル・デザイン



モノリシックカーネルでは、すべてのシステムプロセスがリング0でカーネルの一部として実行されます。

まず、"Monolithic"という言葉について説明しましょう。最初の部分の「Mono」は「1つの」という意味です。後半部分の「lithic」は「石の、または石のような」という意味です。

モノリシックカーネルでは、カーネル全体がリング0のカーネル空間で実行され、コンピュータのリソースやハードウェアに対するより高度なインターフェースを提供します。また、System APIを介して基本的なシステムコールを提供します。

モノリシックカーネルでは、ほとんどの（すべてではありませんが）カーネルサービスは、カーネル自体の一部です。これは、サービスが互いに独立できないという意味ではありません。しかし、ソフトウェアはカーネルの他の部分と非常に緊密に統合されています。このため、モノリシック・カーネルは他の設計に比べて非常に高速で効率的なものとなっています。

すべてのOSサービスは、カーネルの一部として（またはカーネルの拡張として）カーネル空間で実行されるため、デ

バイスドライバーやシステムサービスプログラムに問題があると、システム全体がクラッシュする可能性があります。

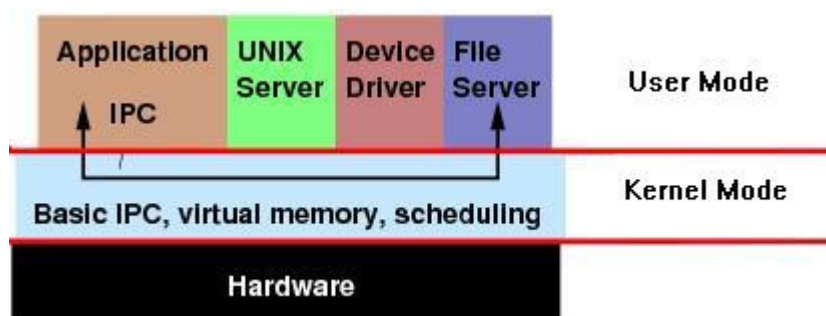
アプリケーションがシステムサービスを要求するときは、**System API**を介してシステムコールを実行します。

例

大規模なオペレーティングシステムでは、以下のようなハイブリッドカーネルが使用されています

- す。Unixライクなカーネル
 - Linux シラブ
 - ル Unix カーネル
 - BSD
 - FreeBSD
 - NetBSD
 - Solaris
 - AI
- DOS
 - DR-DOS
 - MS-DOS
 - Microsoft Windows 9x シリーズ (95、98、Windows 98SE
- 、Me) Mac OS カーネル、～Mac OS 8.6
- OpenVMS
- XTS-400

マイクロカーネルデザイン



マイクロカーネルのデザインでは、カーネルはユーザーモードのシステムサービスに必要な基本的な機能のみを提供します。

マイクロカーネルとは、OSサービスを一切提供せず、サービスを実装するために必要なメカニズムのみを提供するカーネルデザインです。このため、モノリシック・カーネルに比べて、カーネル自体が非常に小さくなります。例えば、マイクロカーネルは低レベルのメモリ管理やスレッド管理、IPC (Inter Process Communication) などを実装します。

上の画像はマイクロカーネルを表示したものです。このカーネルは、基本中の基本しか実装していないことに注目してください。この例では、基本的なプロセス管理とスケジューリング、IPC (Inter Process Communication)、基本的な仮想メモリ管理を実装しています。

カーネルは、(モノリシックカーネルのように) カーネルの一部として組み込まれたものではなく、デバイスドライバやファイルシステムなどの外部のユーザーモードサービスを使用します。このため、外部サービスがクラッシュしても、システムはまだ機能している可能性があり、システムはクラッシュしません。

マイクロカーネルの仕組みを理解するためには、IPC (Inter Process Communication) や、サーバーとデバイスドライバを理解することが重要です。

マイクロカーネルサーバ

マイクロカーネルの「サーバー」は、通常のプログラムにはない特別な権限をカーネルから与えられた外部プログラムです。その特権とは、ハードウェアへの直接アクセスであったり、物理メモリへのアクセスであったりします。これにより、サーバープログラムは、自分が制御しているハードウェアデバイスと直接対話することができます。ちょっと待って、それってデバイス・ドライバーみたいじゃない？そうです。)基本的にはそのようなものです。

マイクロカーネルは非常にミニマルなものであることを覚えておいてください。外部のプログラム、つまりサーバーの助けを借りているのです。

カーネル自体が必要とするサーバーは、通常、カーネルが実行される前にメモリにロードされます。例えば、ファイルシステムを解析するためのコードを格納したファイルシステムサーバーが必要になります。

カーネルにはファイルシステムコードがないため、^ズファイルシステムサーバをロードする手段がありません。そのため、カーネルが実行される前にファイルシステムサーバをロードする必要があります。

どうすればいいのでしょうか？いくつかの方法があります。一つは、カーネルとサポートされるサーバの両方を含む完全なRAMイメージをロードする方法です。もう一つの方法は、起動時にブートローダに必要なサーバをロードし、実行時に何らかの方法でカーネルにサーバの情報を与えるというものです。どちらの場合も、ブートローダはファイルシステムを読み込むコードを決定することができますが、そのコードはそもそもファイルシステムサーバを読み込む必要がなく、ファイルシステムサーバと対話することができます。かっこいいですね。

注：「サーバー」は「デーモン」と呼ばれることもあります。

インタープロセスコミュニケーション (IPC)

マイクロカーネルではIPCが非常に重要です。通常はメッセージを送信しますが、共有メモリを使用することでも実現できます。

プロセスが他のプロセスに「シグナル」を送る方法は数多くあります。マイクロカーネルの接続に関しては、最も一般的に使用されているのがメッセージパッシングであり、最も理解しやすい方法の一つでもあります。

IPCは、サーバーとカーネルが相互に影響しあうことを可能にします。

同期型IPC

同期IPCでは、メッセージを送信するプロセスは、他のプロセスが応答するまで中断されます。他のプロセスがビジー状態の場合、メッセージはキューに格納され、準備ができたときにそのプロセスが処理できるようになっています。

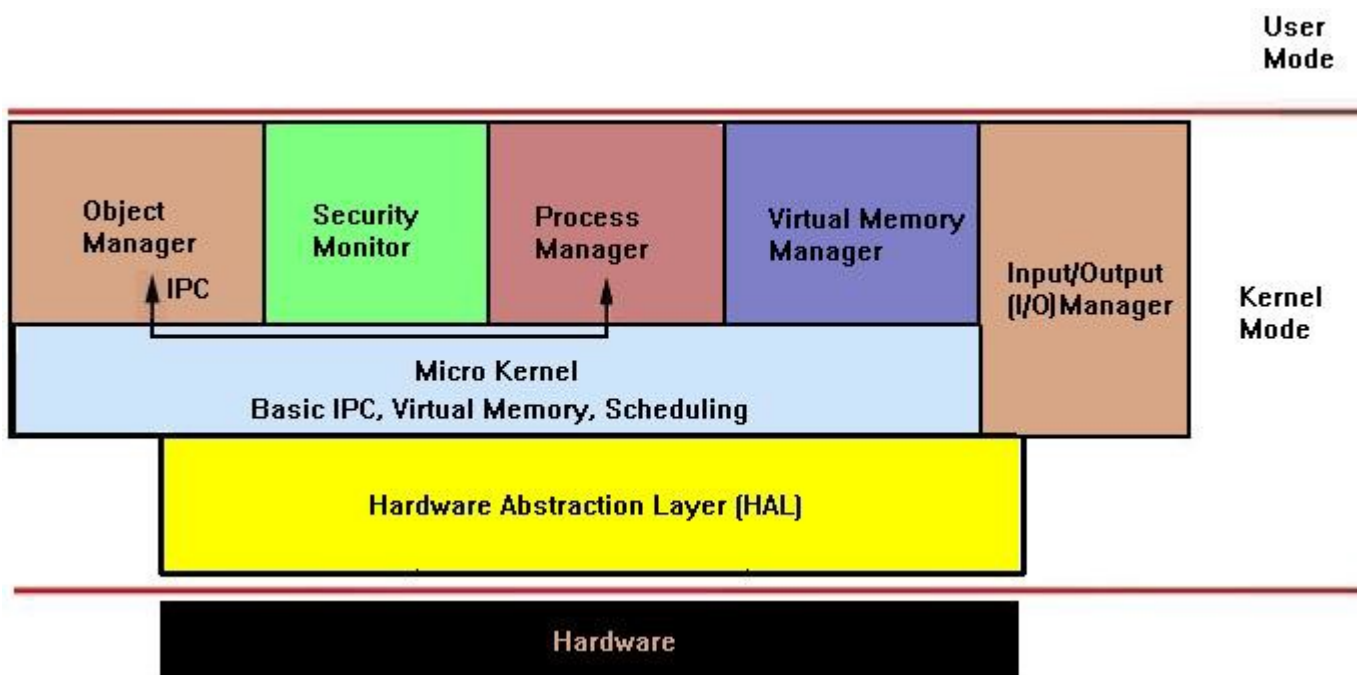
非同期IPC

同期IPCに似ていますが、両方のプロセスが実行され続けます。つまり、プロセスは中断されません。

カーネルデザイン - 概要：二次設計モデル

カーネルの設計方法は無数にあることを覚えておってください。以下は、主要な設計モデル（モノリシックカーネルとマイクロカーネル）をベースにした一般的な設計モデルです。

ハイブリッドカーネル



ハイブリッドカーネルは、モノリシックカーネルの側面を持つマイクロカーネルです。

ハイブリッドカーネルとは、モノリシックカーネルとマイクロカーネルの両方のデザインを組み合わせたカーネルのこと

ハイブリッドカーネルは通常、マイクロカーネルに似た構造を持っていますが、モノリシックカーネルとして実装されています。理解を深めるために、別の見方をしてみましょう。

ハイブリッドカーネルは、マイクロカーネルに似ており、ファイルシステムやデバイスドライバなどに別々のセパレートプログラムを使用します。しかし、モノリシックカーネルのように、これらの分離プログラムはユーザースペースではなく、カーネルの一部として実行されます。

「ハイブリッドカーネル」という言葉を何に適用するかについては、いくつかの議論があります。マイクロカーネル、「マクロカーネル」、あるいは「修正マイクロカーネルまたは修正マクロカーネル」と呼ばれることもあります。ハイブリッドカーネルはそれ自体が設計されているわけではなく、モノリシックカーネルのいくつかの側面を持った修正マイクロカーネルに過ぎません。このため、何をもってハイブリッドカーネルと呼ぶかについては、いくつかの議論があります。

MicrosoftのNTアーキテクチャでは、カーネルの設計モデルにハイブリッドなアプローチを採用しています。マイクロソフトは自社のカーネルを「**Modified microkernel**」と表現しています。

例

大規模なオペレーティングシステムでは、ハイブリッドカーネルを使用しているものがいくつか

- ありますが、これらに限定されるものではありません。BeOS カーネル
 - 俳句カーネル
- BSD
 - DragonFly BSD カーネル
 - XNU カーネル
- NetWare カーネル
- ルプラン 9 カーネル
 - インフェルノ・カーネル
- Windows (NT,2000,2003,XP,Vista) NT カーネル
 - ReactOS カーネル

ナノカーネル

ナノカーネルは、**ピコカーネル**とも呼ばれ、非常に小さなカーネルです。通常であれば、最小のマイクロカーネル構造となります。カーネル自体が非常に小さいため、システム内の基本的なリソースを他のソフトウェアやドライバに頼らなければなりません。

結論

このチュートリアルがそれほど複雑なものではないことは認めざるを得ないでしょう。しかし、私たちがカバーしなければならない非常に重要なトピックの多くをカバーしています。願わくば、このチュートリアルが読者の皆様にカーネルの理解を深めていただき、カーネルが何を担っているかを知っていただきたいと思います。何しろ、これからの章やチュートリアルで構築するのは、このカーネルなのですから。単なるカーネルではなく、基本的なハードウェア抽象化層（HAL）の構築も行います。

この連載では、改良型のマイクロカーネルを開発していきます。これにより、読者はモノリシックとマイクロカーネルの両方の設計を経験し、理解することができます。また、これらのアプローチを混合してハイブリッド・マイクロカーネルを作ることでもあります。実際、私たちのカーネルは、このチュートリアルで紹介されているものと似ています。次のチュートリアルでは、C++を使ってプロセッサの依存性を抽象化するためのHALの基本的な構成要素を開発するとともに、カーネルの完全な設計について触れます。

今後は、複数のコンパイラやプラットフォームに対応するために、複数のバージョンのデモを作る予定です。また、C++とC言語の両方をサポートするようにします。いいでしょ？

次の機会まで。

マイク

BrokenThorn Entertainment 社。現在、DoE と [Neptune Operating System](#) を開発中です。質問やコメントは

ありますか？お気軽に[お問い合わせください](#)。

あなたも記事の改善に貢献したいと思いませんか？もしそうなら、ぜひ[私に教えてください](#)。



第12章

ホーム

第14章





オペレーティングシステム開発 - MSVC++ 2005, 2008, 2010 by Mike, 2008, Updated 2010

このシリーズは、オペレーティングシステムの開発を一から実演し、教えることを目的としています。

はじめに

歓迎します。:)

このチュートリアルでは、Microsoft Visual C++ 2005 を Kernel Land で動作させるための設定について説明します。これにより、素晴らしい MSVC++ 2005 IDE を使用して、完全に作業できるようになります。

今回はエクスプレス・エディションを使用します。プロフェッショナル版をお持ちの方は、スクリーンショットの一部が異なります。ほとんどのオプションは同じですので、ご安心ください。

このチュートリアルは、Visual C++ 2008またはVisual C++ 2010を使用している場合でも問題なく動作します（設定するオプションに若干の変更があります）。

高レベルの言語を動作させるには、作業や設定が難しいものです。さらに、ビルド環境を整えることは、さらに複雑さを増します。

例えば1つ目の問題は、MSVC++はWin32互換のPE実行ファイルとDLLしか出力しないことです。そして、現在の段階では、絶対アドレス1MBにロードされたフラットでピュアなバイナリプログラムしかありません。どうすれば、このフラットな純粋アセンブラプログラムから、本格的なC++プログラムを実行できるのでしょうか？

とはいえ、それは問題の始まりです。C++自体がC++ランタイムライブラリに依存している。それで？**C++ランタイムライブラリは、オペレーティングシステムに依存しています。**私たちはオペレーティングシステムを開発しているので、**C++はオペレーティングシステムの中で動作する標準ライブラリを持っていません。**つまり、標準的なランタイムがない状態で仕事をしなければならないのです。

しかし、待ってください。アプリケーション・ソフトウェアでは、**ランタイム**が私たちのC++環境のためにすべてを初期化する必要があることを覚えていますか（グローバル・コンストラクタの実行、基本的なC++操作のサポート、main()の実行など）。私たちにはランタイムがありませんから、自分たちですべてを行わなければなりません。ランタイムがないのに、どうやってランタイム環境を開発するのかという、鶏と卵の興味深いシナリオが生まれます。

ご覧の通り、C++を正常に**動作**させるのは非常に難しいことです。

このチュートリアルでは、カーネル開発のためにMSVC++ 2005をセットアップし、言語を使いこなせるように設定していきます。また、ランタイムの統合についても見ていきます（つまり、基本的なC++がランタイムに依存しているところを見て、自分たちで構築できるようにします）。

ただし、C++の機能の中には、私たちがまだ実装していない詳細情報に依存しているものもありますのでご注意ください。これらの詳細はかなり高度なものです。例えば、C++の

new と **delete** オペレータを使用するには、作業用の**メモリマネージャ**がすでに存在している必要があります。そのため、このセクションではまだすべてをカバーすることはできません。とはいえ、ここではできるだけ多くのことを設定します。

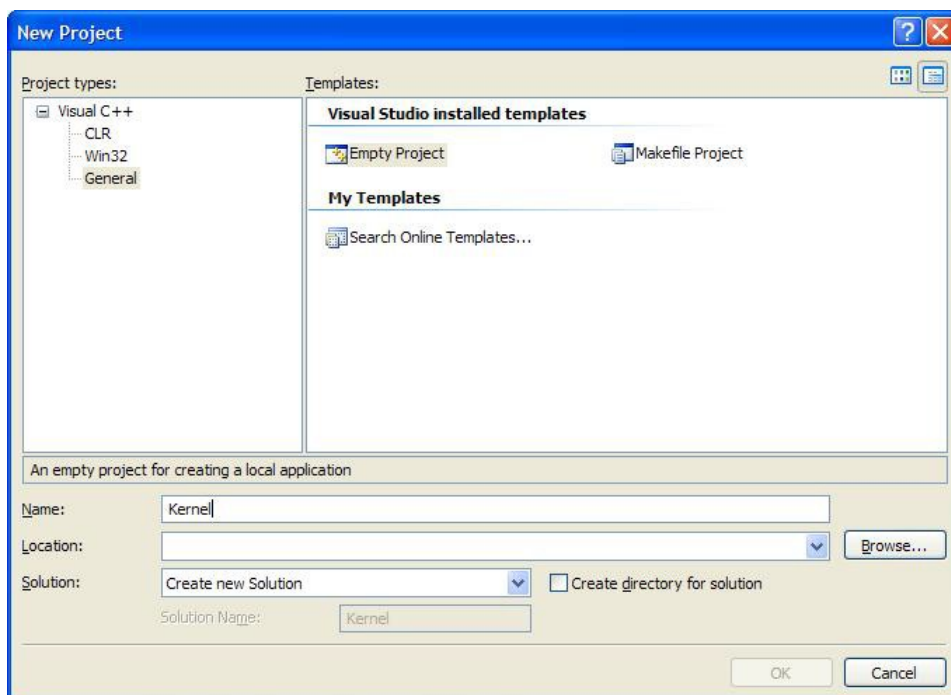
いいですか？

新規プロジェクトの立ち上げ

他のほとんどのプロジェクトと同様に、プロジェクトの作成は非常に簡単です。IDE

で、「ファイル」→「新規作成」→「プロジェクト」を選択します。素敵なダイアロ

グが表示されるはずですよ。



インストールされたテンプレート」で「空のプロジェクト」の設定が選択されていることに注意してください。

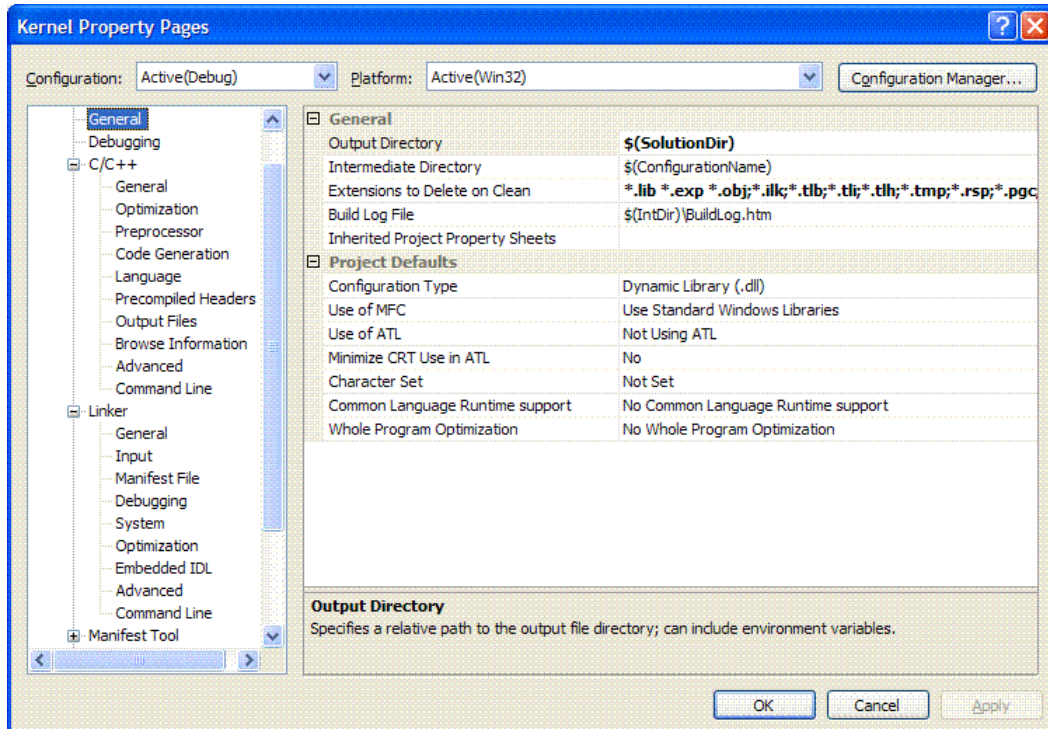
ここで、プロジェクトの名前を入力し、プロジェクトの場所を選択します。完了したら、「OK」ボタンをクリックしてください。これで、新しいプロジェクトが作

成されました。

これは素晴らしいですね。しかし問題は、このコンパイラは私たちが基本的な**Win32**アプリケーションを構築していると仮定していることです。そのため、プロジェクトのすべてのプロパティをデフォルトの設定にし、**Standard C++ Runtime**ライブラリをリンクしますが、これは私たちにとっては機能しません。

そのため、コンフィグレーションの設定を変更する必要があります。

プロジェクトのプロパティにアクセスするには、プロジェクト名を**右クリック**して、「**プロパティ**」を選択します。以下のようなダイアログが表示されます。



すべての設定項目は、上の写真のような形式で説明されています（左の面をご覧ください）。

プロジェクト全体の設定

ほとんどのコンフィギュレーション設定は、構築される環境に完全に依存します。**私たちはまだ非常に低いレベルの環境にいることを忘れないでください。A** コンパイラが生成した**1つ**の間違った命令が、コードをトリプルフォールトにしまいます。

また、**私たちはまだRing 0である**ことを忘れないでください。これは重要なことで、もしコンパイラが生成したコードが現在のオペレーティングシステムの設定で動作しない場合、そのコードはトリプルフォールトになってしまいます。

これは問題です。**設定の多くは、出力コードに影響を与えます。**これらの変更がトリプルフォールトを引き起こすか、あるいは完全に動作するかは、オペレーティングシステムのレイアウトとその設定に依存します。これらの変更の多くは、変更する必要はありません。しかし、特定の設定がトリプルフォールトを引き起こす可能性があるため、**C++コンパイラから「最速で最高のコード」**を出力しようとする場合には、特に注意が必要です。

ただし、変更**が必要**なオプションもあります。ここでは、そのようなオプションをご紹介します。また、その他の有用なオプションについても説明します。これらの具体的なオプションについては、詳しく説明します。

このセクションの最後に、私の現在の設定を掲載します、クール？このようにして、皆さんは自分の設定を比較し、**MSVC++ 2005**の設定について学ぶことができます。

コンフィグレーションタイプ

ここではカーネルを実行ファイルとしてビルドするので、**Application (.exe)** にしておきます。この設定は、「**General**」の下にあります（「**Configuration Type**」の場所は、上のセクションで示した画像を参照してください）。

C++コンフィグ設定

上の画像をもう一度見ると、**C/C++**と**Linker**のプロパティが展開されているのがわかります。**C/C++**のプロパティには、**General**、**Optomization**、**Preprocessor**、**Code Generation**などがあります。ここでは、これらのプロパティについて説明します。

C/C++ > 一般

これらのオプションはいずれも変更する必要はありません。しかし、ここで見ておきたいオプションがいくつかあります。

追加のインクルード・ディレクトリ

このオプションでは、**INCLUDE**ファイルに独自のパスを指定することができます。これにより、独自のファイルをインクルードする際に、そのフォーマットを使用することができます。

```
#include <myheader.h>
```

これは、同じディレクトリ内のファイルのインクルード（**#include "myheader.h"**）と、標準的なカーネルのインクルードディレクトリ内のファイルのインクルード（**#include <myheader.h>**）を区別する際に便利です。

デバッグ情報のフォーマット

MSVC++を使用していますが、**MSVC++**のデバッグ機能は**使用できません**。デバッグは、アプリケーションと一緒に実行する（そしてフックする）ために、ランタイム環境を必要とします。私たちはランタイム環境を持っていないので、この機能を**無効にする**必要があります。

通常、デバッグ情報を追加しても何の問題も発生しません。しかし、今はデバッグを使うことができないので、デバッグ情報を生成する理由はありません。

警告レベル

オペレーティングシステムのコードは非常に複雑になります。わずかな潜在的な問題でも追跡できるようにすることが重要です。このような理由から、私がお勧めするのは**最高レベルに設定してください。**

C/C++ > オプティマイゼーション

最適化

これは任意のオプションに設定することができます。特定の設定でコードがクラッシュする場合は、コードを分解して原因を突き止めてください。このシリーズのすべてのソースコードは、すべてのオプション設定レベルで動作を確認しています。

フレームポインターの省略

これは必須ではありませんが、このオプションを設定すると、ESPが解放されるので、それを利用することができます。

C/C++ > プリプロセッサ

プリプロセッサの定義

ソースコード全体を通して、私はすべてのx86アーキテクチャの依存関係を特別なプリプロセッサ定数の後ろに隠しています。これにより、移植性のないコードを他のアーキテクチャに移植することが容易になります。この定数は **ARCH_X86** です。これは#defineしてもいいのですが、ここにARCH_X86を置く方が簡単です ;)

無視 標準 インクルード パス

もう標準ライブラリはないんですね？)

C/C++ > コード生成

いくつかの選択肢がありますが、そのうちの一つをご紹介します。

C++の例外処理を有効にする

これにはランタイムサポートが必要ですが、私たちにはありません。しかし、近いうちに例外処理を実装する予定です。それまでは、これは「いいえ」に設定してください。

カーネルを書いていると、たくさんのクラスや構造体を使うことになります。これらのほとんどはバイトアラインでなければなりません。ほとんどのコンパイラでは、これらの構造体に余分なパディング（高速化のため）を追加しているため、必要なアライメントが崩れてしまいます。そのため、これを**1バイト (/Zp1)** に設定します。

バッファのセキュリティチェック

この機能を有効にすると、MSVC++ はバッファアンダーおよびオーバーランの可能性をテストする追加コードを追加します。これは、MSVC++ランタイムに依存しており、当社では使用できません。このため、これを使用することはできません。**No (/GS-)に設定してください。**

C/C++ > 言語

ランタイムタイプ情報の有効化

RTTI (Run Time Type Info) にはランタイムのサポートが必要です。ランタイムを無効にしているため、これを使用することはできません。**いいえ (/GR-)** に設定

C/C++ > 上級

ここでの変更は必要ありません。個人的には、**stdcall**よりも**cdecl**を使うことをお勧めします。**cdecl**の方がシンボル名がすっきりしているからです。しかし、それは実際には重要ではありません。

C/C++ > コマンドライン

以下は、私が使用しているコマンドラインです。もし問題があれば、これを参考に見てみてください。この章の最後にあるデモでも、これらのオプションがすべて設定されているのを見ることができます。

```
O2 /Oy /I "...Include\\" /D "ARCH_X86" /X /FD /MT /Zp1 /GS- /GR- /FAs /Fa "Debug\\" /Fo "Debug\\" /Fd "Debug\vc80.pdb" /W4 /nologo /c /Gd /TP /errorReport:prompt
```

このコマンドラインと現在のコマンドラインを比較してみてください。オプションを追加することにした天候によっては、さらにオプションが追加されているかもしれません。

リンカーコンフィグ設定

リンカは、いくつかの理由で私たちにとても重要です。リンカは、コンパイラが生成する最終的なシンボリック名を作成する役割を担っています。これらのシンボリック名は、変数、ルーチン、および定数の数値アドレスを表します。例えば、**"main() "**というルーチンは、**"_main "**というシンボリックネームにコンパイルされます。**アセンブリ言語では、変数やルーチンをシンボリック名で参照します。**このため、C++の**main()**ルーチンを呼び出すには、通常、次のようにします。

```
call     _main; C++のmain() ルーチンの呼び出し
```

リンカは、これらのシンボリック名をすべて含むリンカマップを作成します。これは、デバッグやテストで非常に重要になります。これに伴い、リンカーの設定には必須のものと、オプションのものがあります。オプションの設定は、環境の設定や構成によって動作する場合としない場合があります。

リンカ > 一般

ここで変更が必要なオプションはありません。実際の（または仮定の）フロッピードライブを使用している場合は、個人的には「**出力ファイル**」をフロッピードライブに設定することをお勧めします。こうすることで、最終的なバイナリがフロッピーディスクに格納され、エミュレータでカーネルをすぐにテストすることができます。

リンカ > 入力

その他の依存関係

デフォルトでは、MSVC++ は、kernel32.lib、user32.lib、gdi32.lib、winspool.lib などの多くのライブラリを自動的にリンクします。私たちはこれらのライブラリを使用することはありませんので、これらのライブラリが問題を起こすことはありません。しかし、これらはカーネルに必要な以上の肥大化をもたらします。これを**\$(NOINHERIT)**に設定すると、これらがリンクされないように修正されます。

デフォルトのライブラリを無視する

2021/11/15 13:08

オペレーティングシステム開発シリー

標準ライブラリがないので、**Yes (/NODEFAULTLIB)** に設定してください。^ズ

リンカ > デバッグ

リンカがマップファイルを生成できることを覚えていますか？これにより、すべてのグローバルシンボリックネームの相対的なアドレス位置を確認することができます。これは、私たちがまだバイナリイメージレベルにいることを考えると、非常に重要なことです。これを行うには

- マップファイルの生成を「はい」に設定（/MAP
- Map File Name」には、生成するマップファイルの名前を設定します。
- 適用

リンカ > システム

サブシステム

この値は、プログラムファイルの中に格納されています。この値は、OSにアプリケーションの実行方法を伝えます。本製品はドライバアプリケーションなので、この値をネイティブ (/SUBSYSTEM:NATIVE) です。

ドライバ

このオプションは、このプログラムをカーネルレベルのドライバとしてビルドすることを保証します。これにより、（標準の/FIXEDオプションの代わりに）/FIXED:NOオプションが自動的にエンベッドされ、固定ベース・アドレスの代わりにリロケーション・セクションが生成されます。ここではドライバ・アプリケーションを開発しているので、このオプションをDriver (/DRIVER) に設定します）。

リンカー > 最適化

Refrences to Eleminate unrefrenced data」を設定すると、未参照のシンボル（使用されない変数や関数など）がすべて削除されます。また、「Enable COMDAT folding」を「Remove redundant COMDATS」に設定すると、カーネルのサイズが小さくなり、冗長なCOMDATSの数も減ります。

リンカー > アドバンス

エントリーポイント

これはカーネルのエントリーポイントに設定する必要があります。今回のシステムでは、kernel_entry

ベースアドレス

これは、イメージがロードされるベースアドレスです。カーネルは1MBにロードされることを覚えていますか？このため、0x100000に設定します。

固定ベースアドレス

これはリンカーによって自動的に呼び出されます。Generate a relocation section (/FIXED:NO)に設定してください。

リンカ > コマンドライン

追加オプション

追加オプションのテキストボックスに/ALIGN:512を追加します。これは、適切なセクションアラインメントを確保するために必要です。これを行わないと、カーネルの実行に問題が生じたり、トリプルフォールトが発生したりします。

コマンドライン

これで終わりです。コマンドラインを以下のように比較してみてください。環境の設定によっては、追加のオプションがあるかもしれません。

PE

カーネルの実行

```
OPT: "2_KERNL22-EXE" /INCREMENTAL:NO /NOLOGO /LIBPATH:"...Lib\\"MANIFEST:NO
/NODEFMULTILIB /MAP: "Kernel.map" /SUBSYSTEM:NATIVE /DRIVER /OPT:REF /OPT:ICF /ENTRY: "kernel_entry"
BASE: "0x100000" /FIXED:いいえ /ERRORREPORT:PROMPT
```

ここでは、実行ファイルのフォーマット全体を説明するつもりはありません。プログラマネージャやタスクマネージャの話をするまではね。

しかし問題は、MSVC++はCOFFとPEのファイルフォーマットしか出力できないことです。このため、Stage 2ブートローダの中で解析する方法を見つけなければなりません。

PEのフォーマットについてはまだ詳しく説明するつもりはないので、まずは基本的なフォーマットとコードの仕組みについて説明します。では、早速見ていきましょう

ファイル形式

ファイルイメージをメモリに読み込んだ後は、ディスク上のイメージファイルをそのままコピーしただけです。このため、ファイルを解析するためには、メモリから読み込んだ場所からファイルを読み込めばよいのです。

ファイルフォーマットの解析方法を理解することは非常に重要です。FIRST構造体の最初のバイトは、実際にメモリ上で読み込まれた最初のバイトを表していることを覚えておいてください。



_image_dos_header

この構造は、PEファイル内の最初の構造です。

```
typedef struct _IMAGE_DOS_HEADER { // DOS .EXEのヘッダ
    USHORT          e_magic; // マジックナンバー (MZ
    USHORT          e_cblp; // ファイルの最終ページの
    USHORT          e_cp; // ファイルのページ数
    USHORT          e_crlc; // リロケーション
    USHORT          e_cparhdr; // 段落単位でのヘッダのサイズ
```

```
USHORT e_minallocです // 必要最小限の追加段落
USHORT e_maxallocです // 必要な最大の追加段落
USHORT e_ss。 // 初期（相対）SS値
USHORT e_sp; // SP初期値
USHORT e_csumです。 // チェックサム
USHORT e_ipです。 // IP初期値
USHORT e_csです。 // CS初期値（相対値）
USHORT e_lfarlc; // 再配置テーブルのファイルアドレス
USHORT e_ovno; // オーバーレイ数
USHORT e_res[4]です。 // 予約語
USHORT e_oemidです。 // OEMの識別子(e_oeminfo用)
USHORT e_oeminfoです。 // OEM情報; e_oemid specific
USHORT e_res2[10]です // 予約語
LONG e_lfanew; // 新しいexeヘッダーのファイルアドレス
}image_dos_header, *pimage_dos_header;
```

これについては、完全なPEローダーを作成するまで、まだ完全に理解する必要はありません。今のところ、エントリルーチンのアドレスを探しているだけなので、PEヘッダーの開始部分を含む `_IMAGE_FILE_HEADER` 構造体からエントリルーチンのアドレスを見つける必要があります。

`IMAGE_FILE_HEADER` 構造体のアドレスは、`_IMAGE_DOS_HEADER` の `e_lfanew` メンバの中にあります。そのため、このメンバにアクセスするには、メモリに読み込まれている場所からのバイトオフセットを参照します。

```
mov     ebx, [IMAGE_PMODE_BASE+60].    e_lfanewは、PEヘッダの4バイトオフセットアドレスで、60バイト目にあたります。得ることができます。
add     ebx, IMAGE_PMODE_BASE          ベースの追加
```

イエーイNoe EBX には `_IMAGE_FILE_HEADER` 構造体の開始アドレスが含まれています。これは、PEカーネルイメージが以下の場所にロードされたと仮定しています。
`image_pmode_base`です。

リアルモードDOSスタブプログラム

さて、それでは。もう一度、PEファイルのイメージ構造を見てみましょう（上の図です）。
`_image_dos_header`です。これ、実は便利なプログラムなんです。これは、DOS内からWindowsのプログラムを実行しようとする、「このプログラムはDOSモードでは実行できません」と表示するプログラムです。

実行されるプログラムを変更することができます。これにより、デフォルトのプログラムではなく、独自のプログラムを組み込んで実行することができます。これにはMSVC++の **STUB** コマンドラインオプションを使います。例えば、以下のようになります。

```
/STUB=myprog.exe
```

myprog.exe が 32 ビットのアプリケーションである限り、MSVC++ はそのプログラムを DOS スタブプログラムとして埋め込み、デフォルトのプログラムを使用しません。いいですね？これは、様々な理由で役に立つでしょう。もしかしたら、あなたのプログラムの特別なDOS版を提供することができるかもしれません。

私たちのカーネルはEXEファイルなので、ユーザーがダブルクリックしてWindowsから実行しようとする可能性があります。代わりに、このDOSスタブプログラムが実行されます。かっこいいでしょう？

とにかく、このプログラムのサイズは一定ではないので、それを飛び越えて次のセクションである `_IMAGE_FILE_HEADER` に到達する必要があります。そのため、`_IMAGE_DOS_HEADER` 構造体から `_IMAGE_FILE_HEADER` の位置を取得する必要があります。

_image_file_header

さて...EBXが、この構造体の開始アドレスに入っていることを思い出してください。この構造体は便利ですが、私たちが必要とするものではありません。エントリー・ポイント・ルーチンを実行する方法が必要なのですよね？この構造体のサイズは24バイトで、`_IMAGE_OPTIONAL_HEADER` 構造体はそのすぐ後にあることを知っているの、今はこの構造体をジャンプしてOKできます。

```
USHORT TimeDateStamp
pです。
mov     ebx, [IMAGE_PMODE_BASE+60].    e_lfanewは、PEヘッダの4バイトオフセットアドレスで、60バイト目にあたります。得ることができます
add     ebx, IMAGE_PMODE_BASE
USHORT SizeOfOptionalHeader;
; EBXがOptionalHeaderの先頭を指すようになりました。これを越えて次のセクション( _IMAGE_OPTIONAL_HEADER)にジャンプします
}image_file_header, *pimage_file_header;
```

_image_optional_header

```
struct _IMAGE_OPTIONAL_HEADER
{...
//
// 標準的なフィールドです。
//
USHORT MajorLinkerVersion、
uchar  MinorLinkerVersion、
ulong  SizeOfCode、
ulong  SizeOfInitializedData、
ulong  SizeOfUninitializedData
ulong  、AddressOfEntryPoint、
ulong  BaseOfCode。
ulong  BaseOfDataです。
//
// 追加フィールドです。
//
ULONG ImageBaseで
す
```

```
        ULONGSectionAlign
ment;
        ULONGFileAlignmen
t;
USHORT  MajorOperatingSystemVersion;
USHORT  MinorOperatingSystemVersion;
USHORT  MajorImageVersion。
USHORT  MinorImageVersion;
USHORT  MajorSubsystemVersion;
USHORT  MinorSubsystemVersion;
        ULONGReserved1;
        ULONGSizeOfIma
ge;
        ULONGSizeOfHea
ders;        ULONGChecksum;
USHORT  Subsystem;
USHORT  DllCharacteristics;
        ULONGSizeOfStackRes
erve;
        ULONGSizeOfStackCom
mit;
        ULONGSizeOfHeapRese
rve;
        ULONGSizeOfHeapComm
it;        ULONGLoaderFlags;
        ULONGNumberOfRvaAndSizes
        IMAGE_DATA_DIRECTORY データディレクトリ [IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
}image_optional_header, *pimage_optional_header;
```

これは重要な構造です。名前には「オプション」とありますが、そんなことはありませんのでご安心ください。すべてのPEプログラムに必要な構造です。

ここで重要なのは**AddressOfEntryPoint**というメンバで、これには...ええと...。エントリー・ポイント・ルーチンのアドレスです。例えば...**main()**や**mainCRTStartup()**など、必要に応じて何でも構いません。

EBXがこの構造体の先頭を指していることがわかっているので、あとは**EBX+AddressOfEntryPoint**を参照すればいいのです。その位置から読み取れば、呼び出すべき開始ルーチンの先頭アドレスが得られます。このアドレスを取得したら、あとはその場所にファージャンプするだけで、実質的に**C++**のエントリーポイントと呼び出すことができるのです。

まとめてみる

さて、すべての設定が完了したところで、これをすべてまとめてみま
しょう。画像は**IMAGE_PMODE_BASE**で読み込まれることを覚えて

```
movab    ebx, [IMAGE_PMODE_BASE+60].      ; e_lfanewは60バイト目です。取得する
le
追加     ebx, IMAGE_PMODE_BASE              ; ベースアドレスを追加します。EBXがファイルのsigを指すようにな
                                         った (PE00
```

画像は**PMODE_IMAGE_BASE**にロードされるので、最初の構造体である**_IMAGE_DOS_HEADER**の1バイト目がそこにあります。**DOS_IMAGE_FILE**構造体の**e_lfanew**メンバには、**_IMAGE_FILE_HEADER**のアドレスが格納されていることを覚えておいてください。これは**オフセットアドレス（ベース0を想定）**なので、**メモリにロードした場所にベースアドレスを追加する必要があります**。

```
オプションのヘッダーにジャンプします。
追加     ebx, 24
movab    eax, [ebx].                      IMAGE_FILE_HEADERは、20バイト+シングのサイズ（4バイト）です。
le
追加     ebx, 16                          エントリポイントのアドレスはebxになりました。
```

ここでEBXは**_IMAGE_FILE_HEADER**の先頭を指しています。最初の行はこのセクションを飛び越えています（今は必要ないので）。つまり、**ここでの最初の命令の後、EBXは_IMAGE_OPTIONAL_HEADER構造体の先頭を指しており、ここでAddressOfEntryPointメンバを探し始めます。**このメンバは先頭から16バイトのところにあるので、EBXに16を加えます。

さて、EBXにはエントリーポイントルーチンのアドレスが入っています。しかし、それを呼び出す前に、イメージベースアドレスをエントリーポイントアドレスに追加する必要があります。つまり、エントリーポイントのアドレスは単なるオフセットです。

IMAGE_OPTIONAL_HEADER構造体を振り返ってみると、**ImageBase**メンバが見えてきます。から12バイト（ULONGは4バイト）となっています。**AddressOfEntryPoint**になります。EBXがすでに**AddressOfEntryPoint**を指していることを知っているなので、これはとても簡単です。

```
movab    ebp, dword [ebx].                スタアエントリーポイントアドレス
le
追加     ebx, 12                          ; ImageBase メンバは AddressOfEntryPoint メンバから 12 バイトです。
movab    EAX, DWORD [EBX].               イメージベースを得る
le
追加     ebp, eax                         エントリーポイントのアドレスにイメージベースを追加
```

ebpにエントリーポイントのアドレスが入っているので、それを呼び出します。

```
call ebp; カーネルの実行
```

難しくないでしょう？ここでは、コードセクタ（0x8）を指定する必要がないことに注意してください。理由は、CSにはすでに0x8が含まれているからです。

MSVC++のためのC++ランタイム環境の開発

ご存知のように、**Windowsi**に付属していたランタイムは使えません。その理由はとても簡単です。**C++**の**Windows**ランタイムは、既存の**Windows**オペレーティングシステムに大きく依存しています。私たちは新しいOSを開発しているので、このランタイムは存在しません。

このため、**C++**のランタイムコードを独自に作成する必要があります。これがなかなか厄介なのです。**C++**の機能の多くは、ランタイムの使用を必要とします。しかし、ラ

2021/11/15 13:08

オペレーティングシステム開発シリーズ

ランタイムを無効にしているため、これらの機能を使用すると、コンパイル時に興味深いエラーが発生します。また、単に予測できない場合もあり、トリプルフォールトが発生することもあります。

少し考えてみましょう。アプリケーションにおいて、**main()**を呼び出すのは何でしょうか？ランタイム・ライブラリです。すべてのグローバルオブジェクトを呼び出して初期化するのは？ランタイム・ライブラリです。システムと結びついた特定のキーワードサポート（**new**や**delete**など）を提供するのは？ランタイムライブラリです。**initil**のスタック情報を設定するのは？繰り返します。**ランタイムライブラリです。**

ランタイムライブラリを定義しないと、予想外の結果になることがあります。例えば、グローバルオブジェクトやスタティックオブジェクトが初期化されないことがあります。また、特定のキーワードの使用が予測できないという問題もあります。グローバル・オブジェクトやスタティック・オブジェクトは決して解放されません。また、コンパイル時に特定のルーチンに依存してしまいます。

標準のランタイムでは仮想関数の定義と呼び出しが予測不能になることがあります。純粋な仮想ルーチンの呼び出しは直ちにクラッシュします。そして、**new**、**delete**、**typeid**、**例外**に別れを告げます。

話を短くすると、小さなC++ランタイムを作ることは、C++という言語自体を正しく動作させるために必要不可欠です。

グローバルオペレーター

グローバルな**new**と**delete**の演算子を定義する必要があります。しかし、問題は、私たちにはメモリマネージャがないということです。このため、今のところ、何もしないでください。

これで**new**や**delete**を**new**と**delete**として使用できるようになりましたが、これらは全く何もしていません。

```
void* cdecl operator new[](unsigned int size) { return 0; }
void operator delete(void* p) {}.
void cdecl operator delete[](void* p) {}.
```

純粋な仮想関数呼び出しハンドラ

純粋な仮想関数とは、クラス内で宣言されているものの、定義を持たない関数です。その主な目的は、派生クラスにその関数をオーバーロードさせることです。

純粋な仮想関数を通常の方法で直接呼び出すことはできません。純粋仮想関数を呼び出すと、その関数は実際には存在しない（定義されていない）ため、未定義の動作となります。

純粋仮想関数が何らかの方法で呼び出された場合、コンパイルツールは呼び出しハンドラとして**_purecall()**を使おうとします。これが存在しない場合、結果は予測不可能なものとなります---つまり、トリプルフォールトです。

このため、C++のランタイムで定義する必要があります。

```
int cdecl ::_purecall() { for (;;) return 0; }.
```

通常であれば、このようなことは絶対にあってはならないことなので、このようなことが起きたときには**assert ()**をしたいところです。

浮動小数点サポート

新しい MSVC++ カーネルでは、すべてがうまくいっています。それは、**float i=2/2;** をコンパイルしようとするまでのことですが、**BAM!**エラーが発生しました。具体的には、未解決の外部エラーです...私たちのお気に入りです ;)

これは、定義されていない**new**や**delete**の演算子を使うのと同じで、何の問題もありません。同じように、MSVC++では浮動小数点演算を行うためのルーチンが必要です。

_fltused

これは、MSVC++が浮動小数点が現在使用されているかどうかを判断するために使用されます。C++用にビルドする場合は、これを**1**に設定し、Cリンケージを行う必要があります。

```
extern "C" int _fltused = 1;
```

_ftol2_sse()

最適化レベルに応じて、MSVC++は**float**を**long**に変換するための**_ftol2_sse()**の呼び出しを埋め込むことができます。ここでは**SSE**を使うつもりはないので、**FPU**を使った実装を書きます。

```
extern "C" long declspec (naked) _ftol2_sse() {
    int a;
#ifdef ARCH_X86
    _asm fistp [a]
    _asm mov ebx, a
    _asm retir
#endif
    ed
}
```

その他のルーチン

私は他のルーチンを定義しました。それは**_CIcos()**、**_CIqrt()**、**_CIin()**です。これらのルーチンが必要であることが確認できるまでは、ランタイム・ライブラリの中に入れておきます。

グローバルとスタティックデータの初期化

ここまでは順調ですが、グローバルについてはどうでしょうか？ランタイムは、グローバルに実行されるすべてのルーチンを実行し、すべてのグローバルおよびスタティックオブジェクトを初期化する責任があることを覚えていますか？ランタイムを無効にしているので、それを実行しなければなりません。

そのためには、MSVC++がコンストラクタ (ctor) をどのように処理するかを正確に観察する必要があります。

MSVC++ は、最終的なバイナリイメージ内に **ctors** 用の特別なセクション (**.data**、**.bss**、**.text** などと同様) を使用します。MSVC++ のコンパイラは、起動コードで実行する必要のあるオブジェクトを見つけると、**ダイナミックイニシャライザ**をこのセクション内に配置します。つまり、起動時に実行される必要のあるすべてのダイナミックイニシャライザは、この特別なセクションの中で見つかるということです。

このセクションは、**.CRT** セクションです。これらの動的初期化装置は、**4 バイトの関数ポインタの配列で、.CRT 内に格納されています**。したがって、このセクションを解析する方法が見つかれば、MSVC++ が設定した各関数ポインタを呼び出して、起動時に呼び出される必要のある各ルーチンを呼び出すことができます。

しかし、これらのセクション名はMSVC++特有のものであるため、C++だけではこれを行うことはできません。また、ランタイムなしで構築しているため、.CRTセクションは現在存在しません。このセクションを自分で追加する必要があります。これを行う**唯一**の方法は、プリプロセッサを使用することです。

命名規則

さてさて...このセクションはちょっとわかりにくいかもしれません。MSVC++で使われているセクション名はとても奇妙です。**Serously--.CRT\$XCU?**何を考えているのでしょうか？

2021/11/15 13:08 オペレーティングシステム開発シリー

実は、このセクション名には目的があります。セクション名は、ドル記号「\$」で区切られた2つの部分で構成されています。**最初の部分は、基本的なセクション名として使われます。2番目の部分は、最終的な画像のどこに位置するかを示しています。**

つまり、このような形式のセクション名を考えることができます。

```
.セクション名$ロケーション名
```

セクション名は、.code、.data、.bss、.CRT、その他のセクション名が使用できます。**location_name**は、セクション内のどこにいるかを表すアルファベットの名前で。例えば、**.CRT\$XCA**では、.CRTがセクション名、XCAがそのセクション内の位置です。このロケーション・ネームが何であるかは重要ではなく、重要なのはアルファベット順であるということです。

ここではその一例をご紹介します。

各部分がアルファベットの順に並んでいることに注目してください。2番目の部分の配置は、最終的な画像の中で格納される位置を決定します。上の例では、**.CRT\$XCAが最初、CRT\$XCUが2番目、.CRT\$XCZが最後になります。**

```
.CRT$XCZ
.CRT$XCA
.CRT$XCU
```

別の見方をすると、これらを混ぜ合わせてみると...

ここでも同じことが言えます。**.CRT\$XCAは、再び最初のセクションです。この例ではそれを説明しています。これらのセクションの順序は、そのアルファベットに依存します--aはzの前に来るので、.CRT\$XCAは.CRT\$XCZの前に来ます。**最後の文字に注目してください。難しくないと思いますが、いかがでしょうか?)

これらのセクションを設定すると、リンカーマップの中で見るできるようになります。

新しいセグメント名の作成

新しいセクションを作るためには、**#pragma data_seg()**指令を使う必要があります。この指令は、それ以降に割り当てられたすべてのデータが、この新しいセクション内に配置されることを保証します。

この指令は形を変えています。

```
#pragma data_seg( ["section-name"[, "section-class"] ] )
```

"section_class "は、互換性のためだけに保持されており、MSVC++では無視されます。

重要なのは、最初のパラメータである "section-name "で、これは新しいセクションの名前を与えるものです。

デフォルトでは、**.dataセクションに書き込むことはできません。**このプラグマをパラメータなしで使用します。

```
// ここで割り当てられたすべての変数は、.dataセクションではなく、.CRT$XCAセクション内に配置されます。
// このセクションのデータセグメント(.data)を再度選択してください #pragma data_seg()
```

デフォルトでは、**.CRTセクションへの読み書きはできません。**しかし、**.dataセクションへの読み書きは問題なくできます。**私たちが欲しいのは、**同じのパーミッションは、両方のセクション名に対応しています。**

この問題を解決するには、**2つのセクションを一緒にすることで、両方のセクションに読み書きの能力があることを確認します。**

なるほど、ここでは、次のように考えてみます。セクションに移行して、読み書きの初期化を指示します。プリイメージの.CRT\$XCUセクションに関数ポインタとして格納されています。comment linkの前後にセグメントを宣言すると、命名規則とリンカーのおかげで、それらが互いにすぐ後にあることが保証されます。これらのセクションは隣り合っているため、これらのセクションを指す変数を宣言することで、実質的にインシャライザ配列の最初と最後の関数ポインタを指すことになります。次にこれを見てみましょう...

グローバルの初期化 - セットアップ

実際のコードを見て、それを分解してみましょう。

まず、**読み関数を書き込む型定義に型関数の名前をtypedef**します。この関数ポインタは、各グローバルイニシャライザを指すために使用されます。

```
typedef void ( _cdecl_ *_PVEV)(void);
```

```
/**
 *      MSVC++ は、ルーチンを呼び出すのに役立つ動的初期化子と脱アロケータを作成します。コンパイラとリンカは、すべての動的初期化子を .CRT$XCU と呼ばれるセクション内の関数ポインタ・テーブルにバインドします。
 */
```

```
// 標準C++ランタイム (STD CRT)   xc_a が初期化テーブルの先頭を指しています。
```



```
#pragma data_seg(".CRT$XCA")
_PVFV xc_a[] = { NULL };
```

上記のコードでは、.CRT\$XCAセクションを作成しています。これを "A" で宣言することで、次に定義される.CRTセクションの直前になることが保証されます。

xc_a は、MSVC++ の標準的な CRT 名で、初期化リストの先頭へのポインタとして使用され、.CRT\$XCU に格納されます。

```
//! .CRT$XCUはここにあります。
```

当社の「.CRT\$XCU」は、命名規則により「.CRT\$XCZ」の前、「.CRT\$XCA」の後に配置されることが保証されています。

これは、.CRT\$XCZのセクションです。ここも命名規則に従って、.CRT\$XCU内のイニシャライザ・リストの直後にあることが保証されています。ここで関数ポインタを定義すると、.CRT\$XCU内のイニシャライザ配列内の最後のイニシャライザルーチン (1) を指すことが保証されます。xc_z は MSVC++ CRT で使用される標準的な名前です。

```
_PVFV xc_z[] = { _NULL_ }です。
```

その他のデータは、`data` セクションを複数回呼び出すことで、そのセクションに置き換えてください...

...そして、CRTセクションを `data` セクションに統合し移動して、書き込み可能な CRT セクションにアクセスできるようになります。各ルーチンを初期化する各関数ポインタを `merge1` で呼び出します。警告。ヌルの関数ポインタに注意してください。NULL の関数ポインタを呼び出すと、メモリ内のランダムな場所への無効なジャンプとなり、トリプルフォールトとなります。

環境の浄化

```
decl_initterm ( _PVFV * pfbegin, _PVFV * pfend )
{
```

うれしいですね。これですべてのグローバル初期化ルーチンが実行されました。次は何をするの？もちろん、自分自身の後始末です。

```
while ( pfbegin < pfend )
```

良いことに、これは初期化ルーチンよりもはるかに簡単に作業できます。必要なのは、グローバルなデイニシャライザ関数のポインタの配列を、メモリのどこかに格納する場所を定義することだけです。

```
if ( *pfbegin != NULL )
```

```
((**pfbegin))()のようになっています。
```

これらは、関数ポインタ配列をライブラリ配列のどこにいても追跡するために使用する関数ポインタです。

```
static _PVFV *atexitlist = 0; // 次のイニシャライザに移動する
```

これらの配列がどこにあるかを追跡します。MSVC++ は、各グローバルオブジェクトにデイニシャライザコードを追加し、グローバルデイニシャライザ配列への関数ポインタを連列します。評価され特別に定義されたルーチンである `atexit ()` を呼び出すことによって行われます。

```
static unsigned max_atexitlist_entries = 32;
```

注意：MSVC++ ではこのルーチンが必要です。このルーチンを定義しないと、あらゆる種類の `dtor` を定義する際にエラーが発生します。

```
//!現在のグローバルオブジェクトのルーチンを初期化します。
```

実際の `atexit` は簡単です。グローバルオブジェクトごとに、MSVC++ はこのルーチンを呼び出すコードを埋め込んでいることを覚えておいてください。オブジェクトとしての `dtor` は、パラメータとしてこのルーチンに渡されます。このため、必要なのは `dtor` の配列の最後に追加するだけです。

```
//!グローバルオブジェクトが生成されるたびに、MSVC++ は各 dtor への関数 ptr を用いてこのルーチンを呼び出します。
```

```
{

    // !十分な空き容量があることを確認する
    if (cur_atexitlist_entries >= max_atexitlist_entries)
        return 1;
    else {

        // 終了ルーチンの追加
        *(pf_atexitlist++) = fn;
        cur_atexitlist_entries++;
    }
}
```

```
    }

    0を返す。
}
}
```

dtorsをリストに追加する方法ができたので（MSVC++はこの関数で自動的にこれを行います）、あとは元の関数ポインタ配列を初期化するだけです。

難しいコードではないと思います。

```
void _i386_init(void)
{
    hit(void);
    // 読者にとっては新しい概念がたさくさんあるので、これらはすべてデモの例で説明したほうがいいでしょう。
    // 警告します。通常、STDCはこれを動的に割り当てます。私たちはメモリマネージャを持っていないので、単に
    // とりあえず使うことのないベースアドレス pf_atexitlist
    = ( _PFV *)0x500000;
    // ここまではPEイメージからのエントリーアドレスの取得について説明しました。エントリーポイントのルーチンは2ndステージローダーによって直ちに実行され
    // ます。エントリーポイントをkernel_entryと設定しましたので、それを定義します。
}
```

```
void _cdecl kernel_entry () {...}
```

コードが実行される前に、レジスタとスタックが設定されていることを確認する必要があります。これは、ブートローダの GDT で正しいディスクリプタを参照するために非常に重要です。また、C++では通常スタックを使用しますので、スタックの設定も必要です。

```
#ifdef ARCH_X86
    _asm {

        cli // 割り込みをクリアする--まだ有効にしてはいけない
              10h//データセクタ用のgdttのオフセット0x10を覚えていますか

        mov ax,
        ? mov ds, ax
        mov es, ax
        mov fs, ax
        mov gs, ax
        mov ss,
        ax//ベーススタックの設定
        mov esp, 0x90000
    }
```

次に、現在のスタックフレームポインタを格納します。これにより、呼び出したルーチンの戻り先が確保されます。

では、main()ルーチンを呼び出し、ebp, esp
push ebp
main()を呼び出した後、システムを停止させて、戻ることがないようにします（戻る場所がないので）。

```
#endif
```

これで必要なものはすべて揃いました。ブートローダを実行するがkernel_entryに設定されている限り、このルーチンは開始ベースアドレスに配置されます--IMBに設定されているはずで

```
InitializeConstructors();
```

デモ
//!カーネルのエントリーポイント
main()を呼び出す。

デモダウンロード(MSVC++)
//すべての動的dtorsをクリーンアップするExit(4);

このデモでは、MSVC++ 2005で書かれた32ビットのKernelをロードして実行します。また、このチュートリアルすべてのソースコードも含まれています。

結論
#ifdef ARCH_X86
_asm cli
#endif

やっただけのこのチュートのためのコンテキストの多くは、非常にシンプルですね。MSVC++ 2005を設定し、OSカーネルで使用するコンパイラを使用できるようにしました。また、基本的なC++ランタイム環境の構築、ctorおよびdtorの呼び出し、仮想関数の処理、およびグローバル演算子についても説明しました。

次の数回のチュートリアルでは、さまざまなコンパイラ用の環境を作ることを考えています。このチュートリアルでは、MSVC++ 2005のセットアップについて説明します。

このチュートリアルは書くのが大変で、まだ完成していません。MSVC++には非常に多くのオプションがあり、これらのオプションを詳細に説明するには長い時間がかかります。私は、単なる「これをやる、あれをやる」というオプション設定リストではなく、コンテキストを組み合わせる方法を見つけたかったのです。そのためのフォーマットのスタイルはまだ決めていません。うまくいったのでしょうか？)

2021/11/15 13:08

オペレーティングシステム開発シリーズ

次の機会まで。

マイク

BrokenThorn Entertainment 社。現在、*DoE* と *Neptune Operating System* を開発中です。質問や

コメントはありますか？お気軽に [お問い合わせください](#)。

あなたも記事の改善に貢献したいと思いませんか？もしそうなら、ぜひ [私に教えてください](#)。



オペレーティングシステム開発シリーズ

オペレーティングシステム開発-基本的なCRTとコード設計

by Mike, 2008

このシリーズは、オペレーティングシステムの開発を一から実演し、教えることを目的としています。

はじめに

わーい、わーい。いよいよカーネルとハードウェア抽象化層（HAL）の開発に着手する時が来ました。

前回のチュートリアルでは、基本的なカーネルのコンセプトをまとめ、さまざまな基本的なカーネルデザインのレイアウトを見てきました。また、マイクロカーネルとモノリシックカーネルのデザインから派生したいくつかのコンセプトを使用するため、私たちのオペレーティングシステムのためにハイブリッドカーネルのデザインを開発することにしました。これにより、両方の世界からいくつかのコンセプトを検討することができます。

このチュートリアルでは、カーネル・プログラムの構築と、ハードウェア・アブストラクション・レイヤー・ライブラリの開発を開始します。現時点では、カーネルとハードウェア・アブストラクション・レイヤーを、使用しているコンパイラに応じてCまたはC++といった高レベルのプログラミング言語で開発できるようにシステムを設定しています。

Cコンパイラとの互換性を保つために、C++ではなくCを使用する予定です。しかし、個人的にはCよりもC++の方が好きなので、ソースのC++バージョンを開発するかもしれません：)

では、今週のリストを紹介します。

- グッドコーディングプラクティスの
- 推進 コードのデザインとレイアウト
- データ型の抽象化と基本的な宣言 CRT: _null.h
- CRT: size_t.h
- CRT: ctype.h と ctype
- CRT: va_list.h, stdarg.h/csdarg.h
- デモ。デバッグ用Printfの作成（近日中にアップロード予定）

...これでおしまいです。このチュートリアルでは、HALとカーネルの基本的な設定についてのみ説明しています。

始めましょう。

始める前に...

これがブートローダの世界からの最初の一步です。ブートローダの中では、移植性やシステムへの依存性を気にする必要はありませんでした。結局のところ、ブートローダはその性質上、システムに非常に依存しています。

しかし、今ではブートローダからカーネルへと移行し、CやC++で開発されるようになりました。これは、独自のランタイムライブラリやHAL（Hardware Abstraction Layer）の始まりでもあります--いろいろあるんですね。

しかし、簡単にはいきません。オペレーティングシステムは非常に大きなサイズになります。今回のシステムはどれくらいの大きさになるかわからないので、最初から良いコーディング方法を強調する必要があります。多くの開発プロジェクトは失敗します。しかし、それは複雑すぎるからではありません。どんなプロジェクトでも、正しく設計すれば複雑さを抑えて作ることができます。次はこれを見てみたいと思います...

パンドラの箱

実際のところ、簡単に言えば、コードは悪なのです。コードは非常に無秩序で醜いものです。コードとデザインの無秩序で再帰的な性質のために、さらに複雑さが増してしまうのです。誤解しないでいただきたいのは、まだ多くのコードを書き直す必要があるということです。その理由は、正しいデザインが存在しないからです。これがコードを混沌とさせている原因です。最初の書き込みと書き換えの後、コード自体が非常に不完全な形になっていくことがあります。これはプロジェクト全体、特に大規模なプロジェクトを停止させる傾向があります。というのも、システムの残りの部分は、この醜い書き方や設計の悪いコードの混沌とした性質に頼る必要があるからです。これはまるで大洪水のようなもので、システムのある部分から始まり、ソフトウェアの他の部分へと広がっていきます。

これを防ぐにはどうしたらいいでしょうか。

"パンドラの箱"は悪だと言われていますが、それは間違いです。箱の中に入っていたものが悪だったのだ。箱はただの箱ではないのだ。"- アノニマス

コードが素敵な小さな箱の中に収められている限り、コードの中身がどんなに不格好でも、醜くても構いません。このコードが誰にも見られない素敵な小さな箱の中に収められている限り。箱の中にはデーモンやクリーチャーなどが入っていても構いません。結局のところ、私たちが見ているのは箱だけなのです。それがどのように機能するかを気にする必要はありません。

これこそが、孤立と封じ込めの基本です。つまり、**Encapsulation**であり、ソフトウェアエンジニアリングのほぼ全てのベースとなるものです。

まず、箱の中で何をしているのかを書きます。その後、このモジュールが完成したら、箱を閉じてシステムの他の部分と接続します。しかし、閉じた後の箱は絶対に開けてはいけません。箱を開けてしまうと、箱の中のすべての悪が外に出てしまいます。

はカプセル化を破壊します。一度箱を開けてしまうと、コンパイラエラー、リンカーエラー、ランタイムエラーなど、できる限り多くのコードが感染し、プロジェクト全体が大混乱に陥ります。

しっかりと設計されたシステムでは、すべてのコンポーネントを、互いに接続され、他の大きなボックスの中に入れ子になっているアイソレーションされた（「カプセル化された」）ボックスとして扱います。

カプセル化は、ソフトウェアエンジニアリングにおいて非常に重要な概念です。あなたがオブジェクト指向のプログラマーでなくても、カプセル化の概念は存在します。

インターフェイスとインプリメンテーション

再び「ボンドラの箱」のアナロジーを使うと、「インターフェイス」とは箱のことで、「インプリメンテーション」とは箱の中のことだと言えます。箱のインターフェイス（「パブリック」）部分は、その箱から外の世界へのつながりです。私たちのボックスと、このサブシステム内の他のボックスをつなぐものです。インターフェイス自体には、ボックスが外界に公開するすべての関数プロトタイプ、構造体、クラス、およびその他の定義が含まれており、外界がボックスを使用して対話できるようになっています。これが**インターフェイス**です。このボックスの中にある、モジュールを定義する邪悪なコード、その関数、クラスルーチンなどはすべて、モジュールの**インプリメンテーション**です。

それぞれの箱（コンポーネント）は、シンプルで分かりやすいインターフェイスで構成することが重要です。また、それぞれのコンポーネントが何をやるかが明確でなければなりません。C言語では、グローバルな名前空間は、大量のルーチンで非常に乱雑になります。そのため、これらのルーチンやインターフェイスに名前を付けて、明確に識別できるようにすることが重要です。また、ボックスの実装の詳細（「プライベート」な部分）は、プライベートなメンバーとして保持する必要があります。このような部分をインターフェイスに入れることは、ボックスを開いてしまうことになるのでよくありません（これは悪いことです）。

C言語では、**static**キーワードを使用することで、ルーチンがインプリメンテーションの一部であることを保証することができます。インターフェイスを作るには

extern キーワードを使用します。C++では、**private**、**public**、**protected** キーワードを持つクラスを使うことが推奨されています。

準備

私たちは、大規模なソフトウェアにおける優れたプログラミング手法を促進するために、システムの開発に上記のコンセプトを使用する予定です。

コンパイラ間の移植性を考慮して、C言語を用いてシステムを開発する予定です。ただし、C++を使用しても構いません。

私たちは、拡張性と移植性に主眼を置いています。そのため、ハードウェアに依存する実装はすべて、**Hardware Abstraction Layer (HAL)**という小さな箱の中に隠すことにしています。また、C++スタートアップのランタイムコードはコンパイラに依存するため、**CRT (C++ランタイム) ライブラリ**という小さな箱に入れることにしました。これらはすべて、システムの残りの部分から完全に独立しています。

覚えておってください。大切なのはアイソレーションです。インターフェイスがきれいであれば、どのように隔離されていても構いません。また、一度閉じた箱は決して開けないようにしてください。

そんなことを考えながら、私たちのシステムの第一歩を踏み出してみましょう。

コードのレイアウトとデザイン

このチュートリアルには、これまでで最も複雑なデモが含まれています。そのため、読者の皆様には、デモのソースを開いていただき、チュートリアルに沿って理解を深めていただきたいと思います。

コードデザイン

このシリーズでなぜこの構造を選んだのかを理解することは非常に重要です。第一の理由は**カプセル化**であり、各ディレクトリには個別の**ライブラリモジュール**が含まれています。つまり、これらのモジュールはそれぞれが**ボンドラの箱**なのです。コードの安定性、構造、移植性を維持するためには、これらのモジュールをできるだけ分離しておくことが**非常に**重要です。これを実現するために、私は各モジュールを独立したライブラリモジュールとして扱うことにしました。

2段階のブートローダ（以前のチュートリアルで構築済みです）

SysBoot\

Stage1
and
Stage2

当社のシステムコア

- Stage1のブートストラップ・ローダー
- Stage2 KRNLDR ブートローダ

SysCore\

Debug-Pre-Release Complete Builds
Release-ReverseBuilds
Include_Standard Library Include

directory
Lib's
and
Hal's
and
Kernel's

- 標準ライブラリのランタイム。Crtlib.libまたはCrtlib.dllを出力します。
- Hardware Abstraction Layerの略。Hal.libまたはHal.dllを出力します。
- カーネルプログラム。出力内容 Krnl32.lib または KRNL32.EXE

2021/11/15 13:08

オペレーティングシステム開発シリー

ライブラリモジュールとしてビルドする必要がないのは、**Include/**ディレクトリ内のファイルだけです。これらは単なるヘッダーファイルなので、インプリメンテーションを含む必要はありません。このため、開くべきボックスはありません。

アプリケーションと同様に、私はC++ランタイムコードを最初^ズに実行されるコードとすることにしました。言い換えれば、ブートローダはカーネルを実行しません。その代わりに、ランタイムコード(CRTLIB)を実行して、カーネルの環境を整えてから、カーネルを実行します。

_null.h

イエーイ！！！（笑チュートリアルを始める時が来ましたね。

C++については...

C++をお使いの方は、ライブラリのヘッダファイルについて興味があるかもしれません。つまり、C++では、すべてのCのヘッダーの前に*.hが付加されているのをやめ、**c**が付加されています。つまり、C++では「**#include <stdlib.h>**」の代わりに「**#include <cstdlib>**」を使用します。私たちは、両方の言語で互換性のあるインターフェースを作ることを奨励したいと考えています。しかし、どうすればいいのかと疑問に思われるかもしれません。

実際には、とても簡単です。例えば、**stdlib.h**と**cstdlib**があります。**cstdlib**は、**stdlib.h**を#includeしただけのヘッダーファイルです。私たちのライブラリでも同じことを行います。

これにより、C言語を使用する開発者は**stdlib.h**を使用し、C++を使用する開発者は**cstdlib**を使用することができます。こうすることで、お互いに良い習慣を促すことができます。

トピックに戻る

最初に見てみたいのは、NULLです。ここでは、それほど多くのことを語る必要はありません。しかし、1つだけ細かい点があります。NULLの定義方法は、CとC++のどちらを使用しているかによって異なります。

標準Cでは、NULLは(void*)0と定義されています。C++では、NULLは単なる0です。

```
// NULLの定義解除
#ifdef NULL
# undef NULL
#endif

#ifdef cplusplus
extern "C"
{
#endif
/* 標準的なNULL宣言 */ #define
NULL 0
#ifdef cplusplus
}
#else
/* 標準的なNULL宣言 */
#define NULL(void*)0
#endif
```

このヘッダーには、テンプレートの他にも様々な機能がありますが、これが重要な部分です。他の部分はとても簡単です。

size_t.h

データハイディングについて...

パンドラの箱の理論を思い出してください。箱の中のデータ型は、インプリメンテーションの詳細にあります。いくつかのデータ型は問題ありませんが、いくつかのデータ型は暗黙の了解に留めておいた方が良いでしょう。インプリメンテーションの詳細を維持することで、後方互換性を維持する限り、そのデータ型を使用しているものに影響を与えることなく、そのデータ型について好きなことを変更することができます。

トピックに戻る

この作品については、あまり言うことはありません...

```
#ifdef cplusplus
extern "C"
{
#endif

/* 標準的な size_t タイプ */
typedef unsigned size_t;

#ifdef cplusplus
}
#endif
```

データ型の隠蔽 - stdint.h と cstdint

前節では、インターフェイス内のデータハイディングの重要性を説いていましたが、ポータビリティに関する重要性は説いていませんでした。

それぞれのデータタイプには、指定されたサイズがあります。しかし、各データタイプのサイズは、構築されるコンパイラやシステムに完全に依存しています。このため、データタイプを標準的なインターフェイスの後ろに隠すことが重要になります。

stdint.h

これは約**150**行のかなり大きなファイルです。しかし、どれもそれほど難しいものではありません。このファイルでは、特定のサイズであることが保証されたさまざまな積分データ型が定義されています。

ここでは、システム全体で使用されるファンデメンタルの種類を見ていきましょう。

```
typedef signed          charint8_t;
typedef unsigned        charuint8_t;
typedef
shortint16_t; typedef unsigned
shortuint16_t;         typedef
intint32_t;
typedef                unsignedint32_t;
typedef long            longint64
_t; typedef unsigned long
longuint64_t;
```

32bitシステム用にコンパイルした場合、上記のデータ型は同じであることが保証されます。つまり、**uint8_t**は8ビット、**uint16_t**はWORDのサイズ（2バイト）、というように保証されているのです。データ型のサイズはその名前にエンコードされているので、常にそのサイズを知ることができます。

このファイルには他にも多くのコードがありますが、ほとんどは簡単なものです。

cstdintというファイルは、単に**stdint.h**を**#include**しているだけです。これにより、これらの宣言を2つの方法でインクルードすることができます。

このようにした理由については、**「C++に含まれるものについて」**の項を参照してください。

```
#include <stdint> // C++のみ
```

ctype.h と ctype

ctype.hは、文字列中の文字の種類を判定するためのマクロ群です。これは、標準的な**ASCII文字セット**の異なるプロパティに従うことによって行われます。これは asciitable.com から入手できます。

このヘッダーファイルには、いくつかのマクロや定数が含まれています。

```
extern char _ctype[];

#define
CT_UP0x01/* 大文字 */ #define CT_LOW
0x02/* 小文字 */ #define
CT_DIG 0x04/* 数字 */ #define
CT_CTL 0x08/* 制御 */ #define
CT_PUN 0x10/* 句読点 */
#define CT_WHT 0x20/* ホワイトスペース
(space/cr/lf/tab) */ #define CT_HEX 0x40/* 16進数 */
#define CT_SP0x80/* hard space (0x20) */

#define isalnum(c) (( _ctype + 1)[(符号なし)(c)]の & (ct_up | ct_low | ct_dig))
#define isalpha(c) (( _ctype + 1)[(符号なし)(c)]の & (ct_up | ct_low))
#define iscntrl(c) (( _ctype + 1)[(符号なし)(c)]の & (CT_CTL))
#define isdigit(c) (( _ctype + 1)[(符号なし)(c)]の & (CT_DIG))
#define isgraph(c) (( _ctype + 1)[(符号なし)(c)]の & (ct_pun | ct_up | ct_low | ct_dig))
#define islower(c) (( _ctype + 1)[(符号なし)(c)]の & (CT_LOW))
#define isprint(c) (( _ctype + 1)[(符号なし)(c)]の & (ct_pun | ct_up | ct_low | ct_dig | ct_sp))
#define イスブント(C) (( _ctype + 1)[(符号なし)(c)]の & (CT_PUN))
#define isspace(c) (( _ctype + 1)[(符号なし)(c)]の & (CT_WHT))
#define isupper(c) (( _ctype + 1)[(符号なし)(c)]の & (CT_UP))
#define isxdigit(c) (( _ctype + 1)[(符号なし)(c)]の & (ct_dig | ct_hex))

#define
isascii(c)((unsigned)(c
) <= 0x7F) #define
tolower(c)((unsigned)(c
) & 0x7F)
#define
toupper(c)((unsigned)(c
) & 0x7F)

#define
tolower(c)(isupper(c) ? c + 'a' - 'A' : c)
#define
toupper(c)(islower(c) ? c + 'A' - 'a' : c)
```


ここまではかなりシンプルな内容です。上記のマクロは、個々の文豪を決定したり変更したりするために使用することができます。C++では、**ctype**もありますので、**ctype.h**の代わりに使うこともできます。

va_list.h と stdarg

これらは標準的なヘッダで、変数の引数リストにある無名のパラメータにアクセスするためのマクロを含んでいます。

va_list.h

va_list.hは、可変長のパラメータリストに使われるデータ型を抽象化したものです。

```
/* VAリストのパラメータリスト */
typedef unsigned char *va_list;
```

stdarg.hとcstdarg

これは、これから見る最後の基本ライブラリのインクルードファイルです。CやC++の可変長パラメータリストに使用できる、いくつかの優れたマクロが定義されています。

これらのマクロはかなり複雑なので、1つずつ見ていきましょう。

VA_SIZE

これは、スタック上の幅。VA_SIZEは、スタックにプッシュされたパラメータのサイズを返します。CやC++では、ルーチンにパラメータを渡すためにスタックを使用することを覚えておいてください。32ビットのマシンでは、各スタックアイテムは通常32ビットです。

va_startマクロにプッシュされたオブジェクトの幅を切り上げます。&の前の式は、サイズが0のオブジェクトに対して0を得ることを保証します。*/

```
#define VA_SIZE(TYPE) .....
/* &(LASTARG)は、関数呼び出しの最後の引数の...の前&を指します。
#define va_start(AP, LASTARG) \
    (AP = ((va_list)&(LASTARG)) + VA_SIZE(LASTARG))
```

標準の va_start マクロは 2 つのパラメータを取ります。APはパラメータリスト (va_list型) へのポインタで、LASTARGはパラメータリストの最後のパラメータ (...の直前のパラメータ) です。

このルーチンが行うことは、最後のパラメータのアドレスを取得し、そのアドレスにパラメータサイズのサイズを追加することです。スタックサイズが32であれば、スタック上の最後のパラメータのアドレスに32を追加するだけで、パラメータリストの最初のパラメータの位置になります。

va_end

```
/* va_endには何も無い */
#define va_end(AP)
```

ここではあまり何もすることがありません。

va_arg

```
#define va_arg(AP, TYPE) \
    (ap += va_size(type), *((type *) (ap - \
    va_size(type))))
```

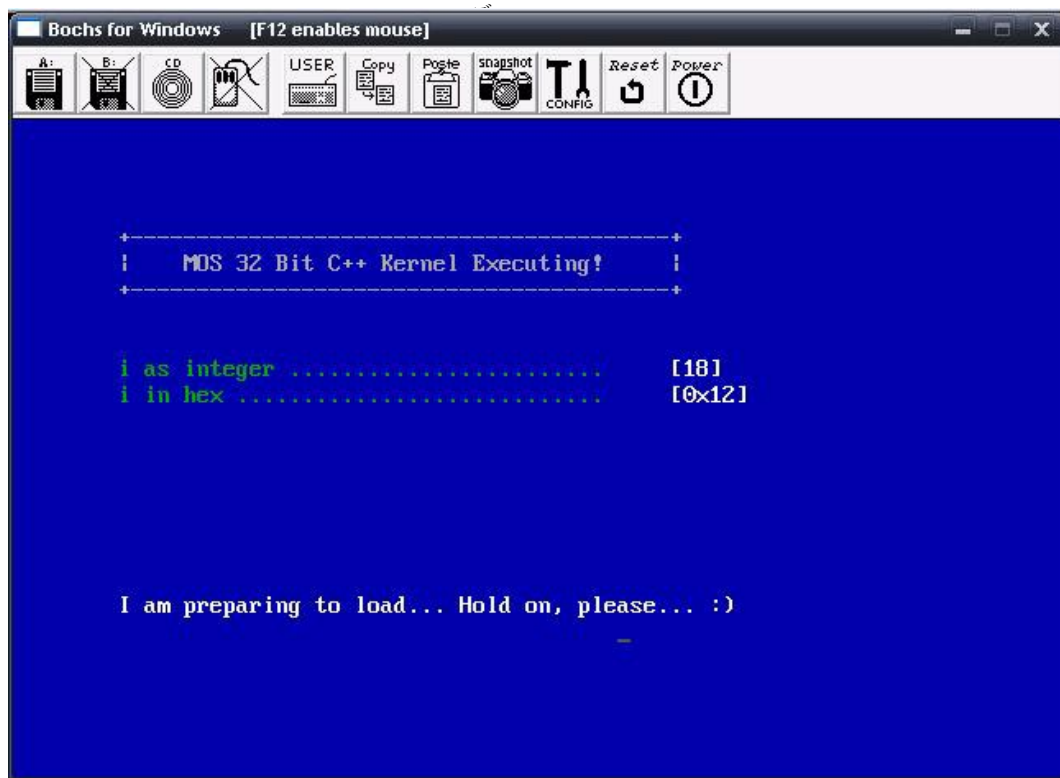
va_arg()は、パラメータリストの次のパラメータを返しますが、これは少し厄介です。AP には、現在作業中のパラメータリストへのポインタを格納します。TYPEには、データ型 (int、charなど) が入ります。

必要なことは、データタイプ (TYPE) のバイト数を可変パラメータリストポインタ (AP) に加えることです。これにより、可変パラメータリストポインタは、リスト内の次のパラメータを指すようになります。

この後、先ほど渡したデータを (ポインタの位置をインクリメントすることで) デリファレンスし、そのデータを返します。

デモ

このチュートリアルにはたくさんのことが書かれています。デモでは、デバッグやテキスト表示に使える独自のprintf()ルーチンを開発しているので、さらに楽しいです。



このデモはかなり複雑です。基本的なC++ライブラリのルーチンと、デバッグ用にテキストを表示する方法を提供したかったのです。このデモでは、すべてのプロジェクトファイルに、ハードウェア抽象化層（HAL）、カーネル、C++ライブラリコードのライブラリが含まれています。言い換えれば...実際よりも複雑に見えるということです :)

[デモダウンロード\(MSVC++\)](#)

結論

このチュートリアルにはたくさんのことが書かれています。読者の皆さんの中には、新しいコンセプトのものもあるかもしれません。

このチュートリアルは、個人的には書きたくありませんでした。コードを書き始める前に、基本的な知識や理論、デザインのコンセプトを説明するための良い方法を見つけたかったのです。基本的な標準ライブラリのヘッダもいくつか見て、システムの基本的な構造も見ています。

これで基本的な必要事項は整いましたので、次のチュートリアルでは実際のカーネルとハードウェア抽象化層（HAL）の構築を始めます。エラーや例外処理の理論と概念、割り込み処理、割り込み記述子テーブル（IDT）、トリプルフォールトを起こさないようにプロセッサの例外をトラップする方法などを説明します。また、スーパー1337のBSODを自作することもできます。）

次の機会まで。

マイク

BrokenThorn Entertainment社。現在、DoEとNeptune Operating Systemを開発中です。質問

やコメントはありますか？お気軽にお問い合わせください。

あなたも記事の改善に貢献したいと思いませんか？もしそうなら、ぜひ私に教えてください。



第13

ホーム

第15





オペレーティングシステム開発シリーズ

オペレーティングシステムの開発 - エラー、例外。 インタラプション by Mike, 2008

このシリーズは、オペレーティングシステムの開発を一から実演し、教えることを目的としています。

注意：このチュートリアルは、ソフトウェア割り込みの処理をカバーしており、ハードウェア割り込みの処理はカバーしていません。ハードウェア割り込みをお探しの方は、**8259A PIC**チュートリアルをご覧ください。ここでは、ハードウェア割り込みを処理するソフトウェア側について説明します。

はじめに

おかえりなさい。:)

前回のチュートリアルでは、システムの基礎知識と設計について説明しました。そう、ここまではかなり基本的で簡単でしたね。次は

いつシステムレベルのコードに入るのかと思っているかもしれません。さて....*ahem* ...お帰りなさい :)

このチュートリアルでは、非常に重要なコンセプトを説明します。**エラー処理**です。エラー処理には、単に問題を処理するだけでなく、問題を**キャッチ**することも含まれます。ここでは、**例外処理**が必要になります。**例外処理**には**割り込み**が必要なので、**割り込み処理**についても説明します。

割り込みはアーキテクチャに依存します。このため、私たちは、**1337**という超高性能でありながら（現時点では）非常に空虚な**ハードウェア抽象化レイヤー**を介して割り込みを管理するためのインターフェースを開発し、カーネルとインターフェースを取り、プロセッサの例外エラーをキャッチするために使用される独自の**トラップゲート**をインストールすることで、完全にハードウェアに依存しないまま、今も昔もトリプルフォールトを防ぐことができるようにします。

楽しそうでしょ？では、早速ご紹介

- しましよう。エラー処理
- 例外処理
- **IRs, IRQs, ISRs**
- ゲートトラップ、インタラプト、タス
- **IDTs**と**IVTs**
- **IDTR**プロセッサレジスタ
- **LIDT**および**SIDT**命令 **FLIH**
- および**SLIH**
- 割り込みの仕組み、スタック、エラーコード カーネ
- ルパニックのエラー画面の開発、つまり**BSOD**

...いろいろなことがあるので、さっそく始めてみましょうか。

エラー、エラー、エラー。

さて、現実を直視しましょう。完璧な人間はいない。コンピュータでは、これがさらに真実となります。私たちはカーネルランドという素晴らしい世界で仕事をしているので、単純なエラーが予測できないソフトウェアやハードウェアの問題を引き起こす可能性があるため、事態はさらに悪化します。

読者の皆さんの中には、すでにトリプルフォールトで経験された方も多いのではないでしょうか。アプリケーションプログラミングでは、ハードウェアを直接扱うことはありません。そのため、エラーになるような問題が少ないのです。しかし、カーネルの世界では事情が少し異なります。トリプルフォールトは、命令やデータのエラーが原因で発生します。プロセッサが解決できない問題が発生した場合、問題が悪化する前にシステムを再起動します。

データの破損、ハードウェアの故障、さらにはシステムの完全な破壊など、問題が深刻化する可能性があるからです。エラー処理の重要性を知ること、これらの問題を解決し、最終的なリリースまで安定したシステムを維持するために重要です。

例外処理

例外処理には**2つの種類**があります。プログラミング言語の構成要素（例えば、標準的なC++の**try/catch/throw**キーワード。コンパイラによっては、**_except**のような追加のキーワードや、**SEH**や**VEH**のようなメカニズムも含まれています）。)私たちが興味を持っているのは、もう一つのプレーヤーです。それは、現在の実行の流れを変える(**?interrupt?**)ように設計されたハードウェアのメカニズムです。この実行の流れを変える条件を例外と呼びます。例外は、エラー（例外）状態を知らせるためにのみ使用されるべきで、通常の動作に使用される条件式には使用されません。

例外が発生すると、実行の流れが変わり、サブルーチン（例外ハンドラ）が実行されます。これにより、サブルーチンがエラー状態を何らかの方法で処理することができます。通常は、ハンドラが呼び出される前に現在の状態が保存されます。これにより、可能であれば、ハンドラが後で実行を継続できるようになります。

例外はハードウェアから設計されていることを忘れてはいけません。つまり、ハードウェアのメカニズムです。これは、ハードウェアの割り込みと、割り込み処理の基本が、関連していることに似ています。

そのため、ハードウェアでの例外処理を理解するには、割り込みに注目する必要があります。次にそれを見てみましょう。

割り込み

割り込みとは、ソフトウェアやハードウェアの注意を必要とする外部の非同期信号のことです。これにより、現在のタスクを中断して、より重要なことを実行することができます。

ハードにはならない。割り込みは、ゼロ除算などの問題をトラップするのに役立つ方法を提供します。プロセッサは、現在実行中のコードに問題があると判断した場合、その問題を解決するために実行する代替コードをプロセッサに提供します。

その他の割り込みは、ソフトウェアをルーチンとしてサービスする方法を提供するために使用されることがあります。これらの割り込みは、システム内の任意のソフトウェアから呼び出すことができます。これは、リング3のアプリケーションがリング0レベルのルーチンを実行する方法を提供するシステムAPIによく使われます。

特に、非同期に状態が変化する可能性のあるハードウェアから情報を受け取る手段として、割り込みは多くの用途があります。

割り込みの種類

割り込みには、「ハードウェア割り込み」と「ソフトウェア割り込み」の2種類があります。**8259A PIC**チュートリアルでは、ハードウェア割り込みについて説明しました。このチュートリアルでは、ソフトウェア割り込みを取り上げます。

ハードウェアインタラプト

ハードウェア割り込みとは、ハードウェアデバイスによって引き起こされる割り込みのことです。通常、これらは注意を必要とするハードウェアデバイスです。ハードウェアインタラプトハンドラは、このハードウェア要求を処理するために必要となります。

このチュートリアルでは、ハードウェア割り込みの処理については、ハードウェア固有のものなので説明しません。**x86**アーキテクチャでは、ハードウェア割り込みは**8259A Programmable Interrupt Controller (PIC)**をプログラミングすることで処理されます。ハードウェア割り込み処理の詳細については、**8259A PIC**チュートリアルをご覧ください。

スプリアスインターラプト

これは、割り込みラインの電氣的干渉や、ハードウェアの不具合によって発生するハードウェア割り込みです。これは絶対に避けたいことです。

ソフトウェアインタラプト

ここからが面白いんですよねー。

ソフトウェア割り込みとは、ソフトウェアで実装され、トリガされる割り込みのことです。通常、プロセッサの命令セットには、ソフトウェア割り込みを処理するための命令が用意されています。**x86**アーキテクチャの場合、これらは通常**INT imm**、**INT 3**です。また、**IRET**、**IRETD**命令も使用します。

INT immと**INT 3**は割り込みを発生させるための命令で、**IRET**クラスの命令は割り込みルーチン（**IR**）から復帰するための命令です。

例えば、ここではソフトウェア命令で割り込みを発生させます。

```
int      3          ソフトウェア割り込み3の発生
```

これらの命令は、ソフトウェアによる割り込みの生成や、割り込みルーチン（**IR**）の実行に使用できます。

ご存知の通り、リアルモードではソフトウェア割り込みが使用できました。しかし、プロテクトモードに移行した途端、**IVT(Interrupt Vector Table)**が無効になってしまいました。このため、割り込みを使用することができません。そのため、割り込みを使うことができず、自分で作らなければなりません。

このチュートリアルでは、ソフトウェアの割り込み処理について説明します。

割り込みルーチン（IR）について

割り込みルーチン(**IR**)は、割り込み要求(**IRQ**)を処理するための特別な機能です。

プロセッサが**INT**などの割り込み命令を実行すると、**IVT (Interrupt Vector Table)**内のその位置で**IR (Interrupt Routine)**が実行されます。

つまり、私たちが定義したルーチンを実行するだけです。難しくないでしょう？この特別なルーチンは、**AX**レジスタの値に基づいて、通常実行する割り込み関数を決定します。これにより、1つのインタラプトコールに複数のファンクションを定義することができます。例えば、**DOS**の**INT21h**関数**0x4c00**のように。

覚えておってください。割り込みを実行すると、自分が作成した割り込みルーチンが単純に実行されます。例えば、**INT 2**という命令は、**IVT**のインデックス**2**の**IR**を実行します。いいですか？

IRは、一般的には**IRQ (Interrupt Requests)**とも呼ばれます。しかし、**ISA**バス内では**IR**の命名規則がまだ使われているので、両方の名称を理解することが重要です。

割り込み要求（IRQ）について

IRQ(Interrupt Request)とは、コントロールバスの**IR**ラインまたは**8259AのPIC(Programmable Interrupt Controller)のIR**ラインを介してシステムに信号を送り、イベントを中断させる行為を指します。

8259 PICが1台のシステムでは、**IRQ**ラインは8本あり、**IR0 IR7**と表示されます。**8259 PIC**を2個搭載したシステムでは、**IRQ**ラインは16本あり、**IR0 IR15**と表示されます。システムの**ISA**バス上では、これらのラインは**IRQ0 IRQ15**と表示されます。

新しいインテルベースのシステムは、コントローラごとに**255**の**IRQ**を可能にする**APIC (Advanced Programmable Interrupt Controller)** デバイ

2021/11/15 13:08

オペレーティングシステム開発シリーズ

スを統合しています。IRQの詳細については、8259A PICチュートリアルまたはAPICチュートリアル of theいずれかを参照してください。

これはどういうことかという、8259A PICはプロセッサのIRライズをアクティブにすることで、ハードウェア・デバイスを介したソフトウェア割り込みコールを生成するようにプロセッサに信号を送り、プロセッサは正しい割り込みハンドラを実行することができるということです。これにより、ハードウェアデバイスの要求をソフトウェアで処理することができます。これについての詳細は、8259A PICチュートリアルを参照してください...これを理解することは非常に重要です。

割り込みサービスルーチン(ISR)

ISR (Interrupt Service Routines) とは、割り込みハンドラのことです。これらは理解するのに重要なので、詳しく見てみましょう。

割り込みハンドラ

割り込みハンドラとは、割り込みやIRQを処理するためのIRのことです。言い換えれば、ハードウェアとソフトウェアの両方の割り込みを処理するために定義するコールバックメソッドです。

ISRには2種類あります。**FLIH**と**SLIH**です

。ファーストレベルインタラプトハンドラ (**FLIH**

FLIHは、デバイスドライバやカーネルの下半分に相当します。これらの割り込みハンドラは、プラットフォームに依存しており、通常はハードウェアの要求に対応し、割り込みルーチン (IR) や割り込み要求 (IRQ) と同様に実行されます。また、実行時間も短いです。また、必要に応じてSLIHのスケジューリングや実行も行います。

セカンドレベルインタラプトハンドラー (**SLIH**

これらの割り込みハンドラは、FLIHよりも長命です。この点では、タスクやプロセスに似ています。SLIHは通常、カーネルプログラム、またはFLIHによって実行、管理されます。

ネストしたインタラプトハンドラー

割り込みハンドラが実行され、割り込みフラグ (IF) がセットされている場合、現在の割り込み中にも割り込みを実行することができます。これを入れ子式割り込みといいます。

リアルモードでのインタラプト

リアルモードの割り込みは、IVT (Interrupt Vector Table) で処理されます。IVT(Interrupt Vector Table)は、インタラプトベクターのリストです。IVTには256個のインタラプトがあります。

IVTマップ

IVTは、物理メモリの最初の1024バイト、アドレス0x0から0x3FFまでに配置されています。IVT内の各エントリは4バイトで、次のような形式になっています。

- バイト0: 割り込みルーチン(IR)のオフセットローアドレス
- イト1: IRのオフセットハイアドレス
- バイト2 : IRのセグメントLowアドレス
- イト3 : IRのセグメントHighアドレス

IVTの各エントリには、単に呼び出すIRのアドレスが含まれていることに注目してください。これにより、メモリ上の任意の場所 (Our IR) に簡単な関数を作成することができます。IVTに関数のアドレスが含まれていれば、すべてがうまくいきます。

では、IVTを見てみましょう。最初の数個の割り込みは予約されており、そのままです。

x86 インタラプトベクターテーブル (IVT)		
ベースアドレス	割り込み番号	説明
0x000	0	0で割る
0x004	1	シングルステップ (デバッグ
0x008	2	ノンマスカブルインタラプト (NMI) 端子
0x00C	3	ブレークポイント (デバッグ
0x010	4	オーバーフロー
0x014	5	バウンドチェック
0x018	6	未定義のオペレーションコード (OPCode) 命令
0x01C	7	コプロセッサなし
0x020	8	ダブルフォールト
0x024	9	コプロセッサ・セグメント・オーバーラン
0x028	10	タスクステートセグメント (TSS) が無効
0x02C	11	セグメントが存在しない
0x030	12	スタックセグメントオーバーラン
0x034	13	GPF (General Protection Fault) について
0x038	14	ページフォルト
0x03C	15	未分類
0x040	16	コプロセッサのエラー
0x044	17	アライメントチェック (486+のみ
0x048	18	マシンチェック (Pentium/586+のみ

0x05C	19-31	予約済みの例外
-------	-------	---------

0x068 - 0x3FF	32-255	ソフトウェアの使用のためのインターラプトフリー
------------------	--------	-------------------------

難しいことはありません。これらの割り込みは、それぞれIVT内のベースアドレスに配置されています。

プロテクトモードでのインタラプト

私たちはプロテクトモードのOSを開発しています。これは私たちにとって重要なことです。ご存知のように、プロテクトモードでは、様々な理由でIVTにアクセスできません。このため、これ以上の割り込みにアクセスしたり使用したりすることはできません。そこで、独自の割り込みを作成する必要があります。

...そして、すべては「インタラプトディスクリプターテーブル」から始まります。

割り込みディスクリプターテーブル(IDT)

IDT (Interrupt Descriptor Table) は、プロセッサがIRを管理するために使用する特別なテーブルです。IDTは、プロセッサのモードに応じて使用されます。IDT自体は256個の記述子の配列で、LDTやGDTと同様のものです。

リアルモード

リアルモードでは、「IDT」は「IVT」とも呼ばれます。詳しくは、上記のセクションのIVTの説明をご覧ください。

プロテクトモード

プロテクトモードでのIDTの動作は、リアルモードとは大きく異なります（これが、プロテクトモードでIVTを使用できない多くの理由のひとつです）。しかし、IVTはまだ使用されています。

IDTは、メモリ上に連続して格納された256個の8バイトのディスクリプターの配列で、IVT内の割り込みベクターによってインデックスされています。次はこのディスクリプターとディスクリプターの種類、そしてIDTの詳細について見ていきます。

割り込み記述子。構造

IDTの記述子は、以下のような形式をとります。フォーマットの一部は、この記述子がどのタイプかによって変わります。

- ビット0...15。
 - **インタラプト／トラップゲート**。オフセットアドレス IRの0-15ビット目
 - **タスクゲート**。
使われていません。
- ビット16...31です。
 - **割り込み／トラップゲート**：セグメントセクタ（使用時0x10
 - **タスクゲート**：TSSセレクト
- タ ビット31～35：未使用
- ビット36...38。
 - **インタラプト／トラップゲート**。予約済み。0でなければなりません。
 - **タスクゲート**：使用していません。
- ビット39...41。
 - **割り込みゲート**。フォーマットは0D110、Dはサイズを表す
 - **01110** - 32ビットディスクリプター
 - **00110** - 16ビットディスクリプター
 - **タスクゲート**。00101でなければなりません。
 - **トラップゲート**。0D111のフォーマットで、Dはサイズを決定します。
 - **01111** - 32ビットディスクリプター
 - **00111** - 16ビットディスクリプター
- ビット42...44：DPL (Descriptor Privilege Level) 。
 - **00**: リング0
 - **01**: リング1
 - **10**: リング2
 - **11**: リング3
- ビット45：セグメントが存在する（1：存在する、0：存在しない）
- ビット46...62。
 - **インタラプト／トラップゲート**：IRアドレスの16～31ビット目
 - **タスクゲート**：使用していま

せん それだけですか？そう、それだけだよ

。)

あとはGDTと同じようにIDTを入力してインストールするだけです。IDTは、IDTよりももっとシンプルなので、さらに簡単です :)上記のリストは、完全な記述子フォーマットです。今のところ、割り込みゲートの開発だけを心配すればいいので、ここではそれだけに集中します。

割り込み記述子。例

GDTと同じように、ビットレベルの例を作成して、すべての動作を正確に説明できるようにします。

まず、割込み記述子の例を見てみましょう。ここでは、すべてを見やすくするために、アセンブリ形式で表示します。

idt_descriptorです。

2021/11/15 13:08

```
.m_baseLow      dw 0
.m_selector     dw 0x8
.m_reserved     db 0
.m_flags        db 010001110b
.m_baseHi       dw 0
```

オペレーティングシステム開発シリーズ

そう、それがディスクリプターのすべてなのだ。それほど難しいことはありませんよね。

これが上の表とどのように関連しているか、分解して各ビットを見てみましょう。

```
00000000 00000000 00000000 00001000 00000000 10001110 00000000 00000000
```

これは記述子ですが、2進法で書かれています。ほとんどの部分はすべて0なので、これは簡単です。

最初の2バイトは、上のコードで示した **m_baseLow** メンバーです。上の表を見ると、これがディスクリプターの最初の16ビットであることがわかります。これは、**割り込みゲートなので、IRのベースアドレスの0〜15ビットを表しています**。つまり、これが私たちのフィールドであれば、IRはアドレス0に位置することになります（IRの位置はさまざまなので、通常はこのようにはなりません）。しかし、今回の例ではこれで大丈夫です）。

次の2バイトは、私たちの **m_selector** フィールドです。これは記述子の16〜31バイト目です。表を見ると、これがセグメントセクタを表していることがわかります。割り込みハンドラにはコードが含まれているので、コードセクタのいずれかを使用する必要があります。これはGDT内のオフセット0x8に定義されているので、これがセグメントセクタになります。

次の数ビットは使用されません。ビット31-35は使用されず、ビット36-38は割り込みゲートのために0でなければならないことがわかります。これが1バイトの大きさで、これが **m_reserved** メンバーとなります。

次のバイトでは面白いことが起こります。文字通り1バイトずつ分解してみましょう。

```
10001110
```

さて、現在は39ビット目です。上の表を見ると、**ビット39-41は0D110でなければならないことがわかります。Dビットがセットされていれば、これは32ビットのディスクリプターです**。これは01110と等しいので、確かに32ビットディスクリプターです。

次の2ビット(上の00)はディスクリプターのバイト42-45で、プライブリッジレベル(DPL)を表しています。00なので、DPLはリング0で実行することになります。

この例の最後の2バイトは、上の表の最後の2バイトです。これはIRのベースアドレスの上位16ビット（この例では0）で、上に表示されている **m_baseHi** メンバーです。

ご覧のとおり、ここではそれほど多くのことをしていません。セクタは常にGDT内のコードセクタのもの（今回は0x8）で、あとはフラグビットとIRベースアドレスを **m_baseLow** と **m_baseHi** に設定するだけです。この後、完全な例を見て、すべてを理解するのに役立ててください。

IDTR プロセッサ・レジスタ

IDTRレジスタは、IDTのベースアドレスを格納するプロセッサ・レジスタです。IDTRレジ

スタのフォーマットは以下のとおりです。

IDTRレジスタ	
ビット16...46 (IDTベースアドレス)	ビット0...15 (IDTRリミット)

簡単でしょう？作成したIDTの**ベースアドレス**がこのレジスタに格納されていることに注目してください。プロセッサはこのレジスタを使って、IDTがどこにあるかを判断しています。

このフォーマットを知ることは非常に重要で、リミットアドレスとベースアドレスの**両方**が含まれています。このため、単純にidtのベースアドレスを与えても動作しません。この問題を解決するには、通常、上記の形式で新しい構造体を作成します。

```
// IDTのレジスタにアクセスするにはどうすればいいのでしょうか？そうですね。
// IDTの新しいアドレスをIDTRレジスタに格納するために使用されます。この命令は、電流保護レベル（CPL）が0（リング0）の場合にのみ使用できます。使用方法是とても簡単です。
// リドート [idt_ptr]
```

LIDT命令 - IDTの読み込み

この命令は、IDTの新しいアドレスをIDTRレジスタに格納するために使用されます。この命令は、電流保護レベル（CPL）が0（リング0）の場合にのみ使用できます。使用方法是とても簡単です。

```
リドート [idt_ptr]
```

これが全てです。idt_baseがIDTのベースアドレスであれば、そのアドレスをIDTRにコピーします。

SIDT命令 - IDTの保存

この命令は、IDTRの値を6バイトのメモリロケーションに格納するために使用されます。この命令は、リング0とリング3の両方のアプリケーションで使用できます。

```
sidt [idt_ptr]
```

インターラプトの仕組み詳細

呼び出すべき割込み手順の発見

割り込みや例外が発生すると、プロセッサは例外番号や割り込み番号をIDTへのインデックスとして使用します。ご存知のように、IDTは上図の形式の256個のディスクリプターの配列にすぎません。IDTR.baseAddress + index * 8という計算を行います。ここで、8はディスクリプターのサイズ（ディスクリプターのサイズは8バイトであることを覚えていませんか？IDTR.baseAddressは、IDTRの上位ビットに格納されているIDTのベースアドレスです。これにより、プロセッサは、割り込みハンドラのディスクリプター・インデックスのベース・アドレスを取得することができます。計算値がIDTR.limitに格納されているIDTリミットサイズよりも大きい場合、IDTのサイズを超えた呼び出しになるため、プロセッサはGPF（General Protection Fault）を実行します。

ディスクリプターは、割り込みゲート、トラップゲート、タスクゲートのいずれかであることを覚えておいてください。インデックスが割り込みゲートまたはトラップゲートを指している場合、プロセッサは例外または割り込みハンドラを呼び出します。これは、コールゲートをCALLするのと同様に行われます。インデックスがタスクゲートを指している場合、プロセッサは、タスクゲートへのCALLと同様に、例外または割り込みハンドラタスクへのタスクスイッチを実行します。

ハンドラの情報やアドレスは、このディスクリプター内に格納されています。プロセッサがスイッチを実行すると

ハンドラの実行

- ハンドラがより低い特権レベル（ディスクリプターのビット42～45）で実行されることになった場合、スタックスイッチが発生します。
 1. ハンドラが使用するスタックのセグメント・セレクトとスタック・ポインタは、現在実行中のタスクのTSSから取得します。プロセッサは、割り込みハンドラのスタック・セグメント・セレクトとスタック・ポインタをこの新しいスタックにプッシュします。
 2. プロセッサは、EFLAGS、CS、EIPの現在の状態を新しいスタックに保存します。
 3. 例外によってエラーコードが保存されている場合、エラーコードはEIPの後に新しいスタックにプッシュされます。
- ハンドラが同じ特権レベルで実行される場合（現在の特権レベル（cpl）が（ディスクリプターのビット42～45）と同じである場合）
 1. プロセッサはEFLAGS、CS、EIPの現在の状態を現在のスタックに保存します。
 2. 例外によってエラーコードが保存された場合、エラーコードはEIP後に現在のスタックにプッシュされます。

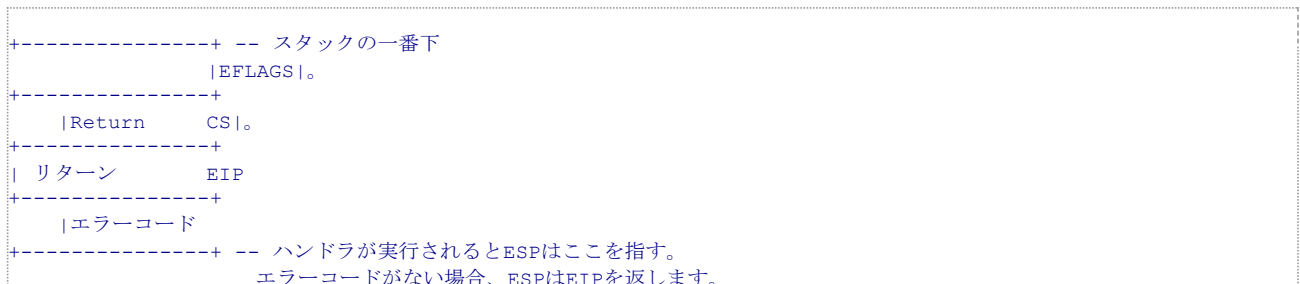
割り込みハンドラが呼び出されたときに、スタックがどのようにプッシュされるか、また、どのような例外でエラーコードがプッシュされるかを知っておくことは非常に重要です。次はこれを見てみましょう。

割り込みハンドラの内部

割り込みハンドラの位置はディスクリプターに格納されているので、プロセッサはハンドラを実行することができます。

ご存知のように、プロセッサが私たちのハンドラを実行するとき、いくつかの追加情報をスタックにプッシュします。私たちのハンドラが私たちと同じリングレベルで実行されている場合（実際にそうなります）、プロセッサはEFLAGS、CS、EIP、およびエラーコードを私たちの現在のスタックにプッシュすることを覚えておかなければなりません。これにより、実行可能な場合は、実行を継続することができます。

これらをまとめると、ハンドラが呼び出されたとき、スタックは次のように設定されます。



この情報は、ハンドラから戻ってきて、例外の原因（エラーコードがある場合）を決定するために使用します。

割り込みハンドラの内部。エラーコードの形式

ハンドラが呼ばれたときに、エラーコードがスタックにプッシュされていれば、その情報をもとにエラーを判断することができます。エラーコードは次のような形式になっています。

- ビット0：外部イベント
 - 0：内部またはソフトウェアのイベントによりエラーが発生した。
 - 1：外部またはハードウェアのイベントによりエラーが発生した。
- ビット1：記述位置
 - 0：エラーコードのインデックス部分はGDTまたは現在のLDTの記述子を参照しています。
 - 1：エラーコードのインデックス部分は、IDTのゲート記述子を参照している。
- ビット2：GDT/LDT。記述子の位置が0の場合のみ使用。
 - 0：エラーコードのインデックス部分が、現在のGDT内の記述子を参照していることを示します。
 - 1：エラーコードのインデックス部分が、LDT内のセグメントまたはゲートディスクリプターを参照していることを示します。
- ビット3-15セグメント・セレクト・インデックス。これは、IDT、GDT、または現在のLDTに、エラー・コードで参照されるセグメント・セレクトまたはゲート・セレクトを示すインデックスです。
- ビット16-31予約

エラーコードは、外部から（INTR、LINT0、LINT1ピンを介して）発生した例外や、INT n命令ではスタックにプッシュされません。

ページフォルト例外エラーでは、エラーコードの形式が異なります。次のセクションでそれを見ていきます。

すべてのハンドラは、**IRET**または**IRETD**命令のいずれかを使用して戻る必要があります。**IRET** は **RET** と似ていますが、保存された **EFLAGS** (ハンドラ実行時にスタックにプッシュされたもの) を復元する点と、**EFLAGS** の **IOPL** フィールドが、現在の保護レベル (**CPL**) が **0** の場合にのみ **0** に設定される点が異なります。 **IF** フラグも、**CPL** が **IOPL** 以下の場合にのみ変更されます。

ハンドラ実行時にスタックスイッチが発生した場合、**IRET**は中断された手続きのスタックにもスイッチバックします。

x86の例外

例外あり。リスティング

すべての例外は、**IVT**または**IDT**内の最初の数個の割り込みとして定義されています。以下は、**x86**クラスのプロセッサから生成された例外の完全なリストです。

- **フォールト** - リターンアドレス (ハンドラが呼び出されたときにスタックにプッシュされたリターン**CS:EIP**。詳細は**割り込みハンドラの内部**を参照してください) は、例外の原因となった命令を指しています。例外ハンドラは、問題を解決してからプログラムを再起動し、何事もなかったかのように見せることがあります。
- **Trap** - リターンアドレスは、完了したばかりの命令の後の命令を指します。
- **Abort** - リターンアドレスは常に確実に供給されるとは限りません。アボートを起こしたプログラムは、決して継続されることはありません。

x86プロセッサの例外			
割り込み番号	クラス	説明	エラーコード
0	フォールト	0で割る	なし
1	トラップまたはフォールト	シングルステップ (デバグガ	なし。デバグレジスターから取得可能
2	未分類	ノンマスカブルインタラプト (NMI) 端 子	該当なし
3	トラップ	ブレークポイント (デバグガ	なし
4	トラップ	オーバーフロー	なし
5	フォールト	バウンドチェック	なし
6	フォールト	無効なOPCコード	なし
7	フォールト	デバイスが利用できない	なし
8	アボート	ダブルフォールト	常に0
9	アボート (予約済み、使用しないでください	コプロセッサ・セグメント・オーバーラン	なし
10	フォールト	タスクステートセグメント (TSS) が無効	以下のエラーコードを参照してください。
11	フォールト	セグメントが存在しない	以下のエラーコードを参照してください。
12	フォールト	スタックフォールト例外	以下のエラーコードを参照してください。
13	フォールト	GPF (General Protection Fault) につ いて	以下のエラーコードを参照してください。
14	フォールト	ページフォルト	以下のエラーコードを参照してください。
15	-	未分類	-
16	フォールト	x87 FPUエラー	なし。x87 FPUは独自のエラー情報を提供します。
17	フォールト	アライメントチェック (486+のみ	常に0
18	アボート	マシンチェック (Pentium/586+のみ	なし。MSRから得られるエラー情報
19	フォールト	SIMD FPUの例外	なし
20-31	-	予約	-
32-255	-	ソフトウェアでの使用が可能	該当なし

IRQ 0とシステムタイマー

ご存知のように、プロテクトモードに入る場合、すべての割り込みを無効にする必要があります。これをしていないと、次のクロックティックですぐにシステムがトリプルフォールトになってしまいます。これはなぜか？

システムタイマ (通常は**8253プログラマブルインターバルタイマ (PIT)** の一形態) は、**IRQ 0**を使用してクロックティックが発生したことを知らせます。このデバイスは、システム**BIOS**によってこのように構成されています。

でも、待ってください。上の表を見ると、「**Divide by 0**」のエラーが出ていませんか？ **ドンゴです**。

プロテクトモードに切り替えたことでテーブルが無効になったため、この先どうなるかわかりません。このため、次のシステムティックでは直ちにトリプルフォールトとなり、切り替え前に割り込みを無効にしなければならない理由となります。

また、**8253プログラマブルインターバルタイマ(PIT)**は、**ハードウェアデバイス**であることにも注意が必要です。上の表を見て、例外 (**IRQ 0**) が発生していることに気がつくますか？実際にエラーが発生したのか、それともただのダニなのか、どうやって判断するのでしょうか？

もう少し詳しく見てみましょう。

8259Aプログラマブルインタラプトコントローラ (PIC) のリマップ

8259A PICは、ハードウェア割り込みを制御するための標準的なコントローラです。ハードウェアマイクロコントローラは、**PIC**に接続されたそれぞれの**IR**ラインで**PIC**に信号を送ります。これにより、**PIC**はハードウェアデバイスが注意を必要としていることを「知る」ことができ、デバイスの要求を処理するために割り込みを発生させるようにプロセッサに信号を送ります。

上記の例では、**8253 PIT**が**8259A PIC**にシステムチックを処理するように信号を送っていたため、**IRQ 0** (**8253 PIT**は**IRQ 0**を使用していることを

2021/11/15 13:08

オペレーティングシステム開発シリーズ

覚えておいてください) が発火し、a) 0による除算の例外でもあり、~~b)~~ まだ書いていないので無効なコードでもあるというトリプルフォールトが発生しました。

この問題を解決するには、8259A PICマイクロコントローラを再プログラムして、ハードウェアデバイスが異なるIRQを使用するように再マッピングする必要があります。

IFはハードウェア割り込みにのみ適用されるため、IFが0（割り込み禁止）であってもソフトウェア割り込みを使用することができるとに留意してください。ただし、ハードウェア割り込みを再び有効にしたい場合は、PICを再プログラムする必要があります。

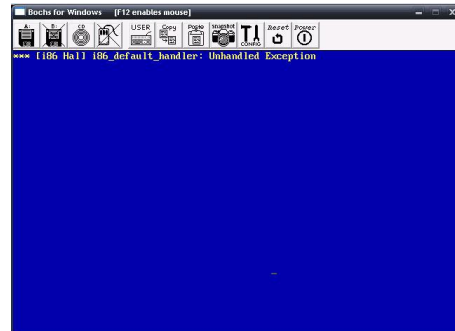
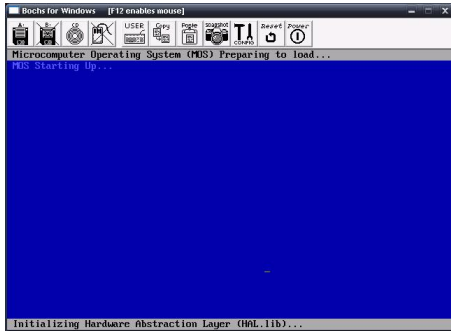
8259A PICは、プログラミングがかなり複雑なマイクロコントローラです。幸いなことに、そのモードのほとんどは我々には当てはまらない。

最後のデモでは、PICを再プログラムし、割り込みを再度有効にしています。このチュートリアルを完全に理解するためには、[8259Aプログラマブル割り込みコントローラのチュートリアル](#)を読むことをお勧めします。

デモ

デモの結論に戻る

注：これらの画像は、画面に合わせて縮小されています。



最初のスクリーンショットは、カーネルがHALを初期化している様子を示しています。2枚目のスクリーンショットは、割り込みが発生したときの様子を示しています。我々のデフォルトハンドラがどのように割り込みをキャッチするか注目してください。

このデモは、このチュートリアルで多くの内容をカバーしているため、かなり複雑です。

このデモでは、カーネルが使用する新しいグローバルディスクリプターテーブル（GDT）と、インタラプトディスクリプターテーブル（IDT）をインストールします。また、ソフトウェア割り込みを処理するための優れたインターフェイスを作成します。なお、ここではハードウェア割り込みは扱っていません。次のチュートリアルでは、**8259A PIC** と **8253 PIT** マイクロコントローラのインターフェイスを HAL に追加する予定です。これにより、ハードウェア割り込みをキャッチしたり、ハードウェア割り込みを有効にしたり、システムタイマーを提供したりすることができるようになります。楽しみですな :))

デモをもう少し詳しく見て、すべての動作を確認してみましょう。

ハードウェアの抽象化

このデモには、今まで見たことのない多くの追加ファイルが含まれています。このため、コードダンプのようになってしまいましたが、これは避けたいことです。その多くは非常にシンプルなもの、私たちが見てきたものであり、ブートローダにも実装されています。**idt.h**や**idt.cpp**の中には、初めて見るものもありますが、ここで学んだことをカバーしています。**割り込み記述子テーブル(IDT)**です。

これは、**HAL (Hardware Abstraction Layer)** の始まりでもあります。

ご存知のように、私はこの連載を始めるにあたり、ハードウェアの抽象化とその重要性を強調してきました。その理由は、Halの開発を続けているうちにすぐにわかるでしょう。ここでは、HALをカーネルから完全に独立させておくことの利点もわかるかもしれません。

それでは、HALの主要なインターフェイスの始まりをご紹介します。

Hal - include/hal.h - プラットフォームに依存しないHAL用インターフェイス

これは、HALとカーネルの間のインターフェイスです。これは標準のインクルードディレクトリの一部であり、インプリメンテーションとは完全に分離しています。すべてのルーチンは **extern** と宣言されています。これは、ヘッダファイルが、その中のルーチンを定義しているすべてのインプリメンテーションによって使用されることを意図しているからです。暗黙の了解はアーキテクチャに依存しますが、インターフェイスは特定の暗黙の了解に結合されることはなく、完全にハードウェアに依存しないものとなっています。

暗黙の了解自体はアーキテクチャに依存していますが、異なるアーキテクチャのために暗黙の了解を単純に構築することができます。各実装は共通のインターフェイスを使用しており、ダイナミックローディングをサポートしているので（**hal.dll**のように）、**a)**異なるアーキテクチャ用に構築する際に使用する静的なHal実装をリンクする、**b)**異なるHalを独立して構築し、起動時にどのHALを使用するかを選択する、のいずれかを行うことができます。これらのHALはすべて同じインターフェイス（**Hal.h**）を使用しているので、異なるインプリメンテーション（つまり異なるハードウェアセットアップ）を使用するためにカーネルを変更する必要はありません。

現在、その中には2つの機能しかありません。必要に応じて追加していきます。

おそれなく、これらの初期化メソッドの引数タイプを変更して、スタートアップとシャットダウンのパラメータを設定できるようにしたいと思います。いずれにせよ、**init**は非常に一般的なルーチンであり、実装のために必要であれば、ハードウェアのセットアップとシャットダウンを行う方法を提供することを目的とします；

halの中には、**gdt**、**idt**、**cpu**用の非常にシンプルなソフトウェアのレイヤーがいくつかあり、**hal.cpp**があります。これらは下の層を初期化するだけなので（**Hal.cpp**は**cpu**の初期化ルーチン呼び出し、それが**gdt**と**idt**の初期化メソッド呼び出し）、このチュートリアルに必要な以上の複雑さを加える可能性があるため、ここには掲載しません。

ここでは、**gdt**セットアップコード、**idt**セットアップコード、**kenrel**の**main()**ルーチンに注目してみましょう。いいですか？

ここでは、GDTの詳細を説明しません。GDTの詳細については、[チュートリアル8](#)を参照してください。

Hal - hal/gdt.h - グローバルディスクリプターテーブル

ディスクリプタ・テーブル...再び!

そう、GDTがあなたを悩ませているのです!!!...そう、YOU!!!

それにしても、GDTはなかなか複雑な構造をしていますね。ご存知の通り、GDTは記述子の配列です。GDTの記述子のフォーマットは何でしたっけ？そうか、それならば....。

- **56～63ビット目** ベースアドレスの24～32ビット目
- **ビット55** : グラニュラリティ
 - **0** : なし
 - **1** : リミットが4K倍になります。
- ビット54** : セグメントタイプ
 - **0** : 16ビット
 - **1** : 32ビット
- **ビット53** : 予約済み-0にすべき
- **ビット52** : OS用の予約
- **ビット48～51** : セグメントリミットのビット16～19
- **ビット47** : セグメントがメモリ内にある（仮想メモリで使用）
- **ビット45-46** : ディスクリプタの特権レベル
 - **0** : (Ring 0) Highest
 - **3** : (Ring 3) Lowest
- **ビット44** : ディスクリプタービット
 - **0** : システム記述子
 - **1** : コードまたはデータ記述子
- **ビット41-43** : ディスクリプタ・タイプ
 - **ビット43** : 実行可能セグメント
 - **0** : データセグメント
 - **1** : コードセグメント
 - **ビット42** : 拡張方向（データセグメント）、準拠（コードセグメント）
 - **ビット41** : 読み出し可能、書き込み可能
 - **0** : 読み出しのみ（データセグメント）、実行のみ（コードセグメント）
 - **1** : 読み取りと書き込み（データセグメント）、読み取りと実行（コードセグメント）
- **ビット40** : アクセスビット（仮想記憶で使用） **ビット16～39** : ベースアドレスの0-23ビット
- **23ビット** **ビット0-15** : セグメントリミットの0-15ビット

恐怖で悲鳴を上げながら走り去ります。

わかった、わかった、もうやめよう :) しかし、真剣に考えれば、インテルはもっと素敵な構造にすることができたと思いませんか?:)

C構造の構築

この構造を、C言語の組み込み型を使ったC言語スタイルの構造の後ろに隠すことができます。最初の15ビットがセグメントの限界（uint16_tのサイズ）であることを知っているのも、**それがデータメンバー1です**。次の16ビットはベースアドレスの0-23ビットで、これは1つのuint16_tまたは2つのuint8_tとして表現できます。これが**データメンバー2と3です**。次の16ビット（GDTの41～56ビット）は16ビットです。これは、フラグ値を含む醜い構造体の大部分であり、もちろん、2つのuint8_tまたは1つのuint16_tを使用して表現することができます。これが**次のデータメンバーです**。最後のバイトはベースアドレスです。これが**最後のデータメンバーです**。

上記を見ると、その醜い構造は、構造の中の4～5人の素敵なメンバーで表現することができます。これが私たちの構造です。この構造体を上記の説明や表と比較して、すべてがどこに収まっているかを確認してみてください。また、この構造体は1バイトにパックされているので、64ビットの大きさが保証されていることも覚えておいてください。

```
#ifndef _MSC_VER
#pragma pack (push,
1) #endif

//! gdtの記述子gdt ディスクリプタは、特定の
//! メモリブロックとパーミッション。

struct gdt_descriptor {

    //! セグメントリミットのビット0～
    15 uint16_
                                tlimit
;

    //! ベースアドレスのビット0～23
    uint16_
                                tbaseLo
;

                                uint8_tbaseMid;

    //! ディスクリプタのビットフラグ。 uint16_tflags以上のビ
    ットマスクを使って設定します。

    //! ベースアドレスのビット24～32
    uint8_
                                tbaseHi
;
};
```

簡単ですね。構造体の中でフラグバイトを構築するために、多くのビットフラグを設定することができます。ヘッダファイルを見て、それらがどのように動作するかを確認してください。基本的には、設定したいビットフラグのビット和をとります。これは次のセクションで説明します。

gdtの抽象化

チュートリアル8では、プロテクトモード、gdt、gdttrについて説明しました。gdttrは、使用するGDTを指定するためのプロセッサ内部のレジスタです。48ビットのポインターで、以下のフォーマットに従わなければなりません。

- **ビット0-15:** gdt全体のサイズ
- **ビット16-48:** gdtのベース・アドレス

さて...これはC言語の構造体に変換するのが簡単です。上のフォーマットに沿っていることに注目してください。

```
#ifndef _MSC_VER
#pragma pack (push,
1) #endif

// ! プロセッサのgdttrレジスタはgdtのベースを指します。これにより
// ! ポインターを設定します
struct gdttr {

    // ! gdtのサイズ
    uint16_t          m_limit
                        です。
    // ! gdtのベースアドレス
    uint32_t          tm_base
                        ;
};

#ifdef _MSC_VER
#pragma pack (pop,
1) #endif

グローバルディスクリプターテーブル
(GDT) static struct
gdt_descriptorの静的
構造体gdttr
                                _gdt [MAX_DESCRIPTOR];
                                _gdttr;
```

このページでは、新しいGDTと_gdttrが紹介されています。これらは、プロセッサのGDTRレジスタを設定する際に参照されます。

gdt_install():gdtをgdttrにインストールします。

このルーチンは非常にシンプルなものです。やっていることは、lgdt命令を使ってGDTRをgdttrポインタでロードするだけです。CSが変化しないように、ここではフェージャンプをする必要はありません。

```
// ! gdttrをインストールする
static void gdt_install ()
{
#ifdef _MSC_VER
    _asm lgdt [_gdttr] です。
#endif
}
```

gdt_set_descriptor():gdtに新しい記述子を設定します。

このルーチンは、GDTに新しい記述子をインストールするために使用されます。ほとんどの場合、それほど難しくはありません。醜いコードは、フラグの設定を行うときです。

```
// ! グローバルディスクリプターテーブルへのディスクリプターの設定
void gdt_set_descriptor(uint32_t i, uint64_t base, uint64_t limit, uint8_t access, uint8_t grand)
{
    if (i > MAX_DESCRIPTOR)
        を返すことができます。

    // ! ディスクリプターをヌル化する
    memset ((void*)&_gdt[i], 0, sizeof (gdt_descriptor));

    // ! リミットとベースアドレスの設定
    _gdt[i].baseLo = base & 0xffff;
    _gdt[i].baseMid = (base >> 16) & 0xffとなります。
    _gdt[i].baseHi = (base >> 24) & 0xff となります。
    _gdt[i].limit= limit & 0xffff;

    // ! フラグと壮大なバイトを設定
    _gdt[i].flags = access;
    _gdt[i].grand = (limit >> 16) & 0x0f;
    _gdt[i].grand |= grand & 0xf0;
}
```

i86_gdt_initialize() - gdtを初期化する

これですべてがまとまります。GDTRの構造を設定し、GDTにいくつかのデフォルトディスクリプターをインストールし、最後にGDTをインストールするだけです。簡単に説明すると、このGDTは、ブートローダに使用したものと同じものです。ベースアドレスは0、リミット（最大アドレス）は4GB(0xffffffff)です。フラグはすべて **gdt.h** で定義されています。これらは可読性を高め、醜いマジックナンバーを取り除くために定義されています。フラグがあることで、ディスクリプターが何のためにあるのかがよりわかりやすくなるはずです。

```
gdtの初期化
int i86_gdt_initialize () {

    // ! gdtrの設定
    _gdtr.m_limit = (sizeof (struct gdt_descriptor) * MAX_DESCRIPTOR)-1;
    _gdtr.m_base = (uint32_t)&_gdt[0]. となります。

    スルの記述子を設定 gdt_set_descriptor(0, 0, 0,
    0, 0);

    // ! デフォルトのコード記述子を設定
    gdt_set_descriptor (1,0,0xffffffff,
        i86_gdt_desc_readwrite|i86_gdt_desc_exec_code|i86_gdt_desc_codedata|i86_gdt_desc_memory,
        i86_gdt_grand_4k | i86_gdt_grand_32bit | i86_gdt_grand_limithi_mask) 。)

    // ! デフォルトのデータ記述子を設定
    gdt_set_descriptor (2,0,0xffffffff),
        i86_gdt_desc_readwrite|i86_gdt_desc_codedata|i86_gdt_desc_memory,
        i86_gdt_grand_4k | i86_gdt_grand_32bit | i86_gdt_grand_limithi_mask) 。)

    // ! gdtrのインス
    トール
    gdt_install ();

    0を返す。
}
```

Hal: Interrupt Descriptor Table

ここからが面白いところです。IDTのインターフェースは、idt.hとidt.cppのソースファイルに含まれています。

hal.h - idt_descriptor

これが割り込み記述子の構造です。このフォーマットと、このチュートリアルで見たディスクリプターのフォーマットを比較してみると、まったく同じフォーマットになっていることに気づくはずです。

それぞれのメンバーが何を表しているのか、そして割り込みディスクリプターのどこにあるのかを見てみましょう。

- **baseLo** - 割り込みルーチン（IR）のベースアドレスの最初の16ビットです。
 - これは、全体の割り込み記述子の中のビット0～15です。 **割り込みディスクリプター** に掲載されている表と比較してください。
- **baseHi** - IRのベースアドレスの16-31ビット目
 - これは、全体の割り込みディスクリプター内のビット16～31です。
- **予約済み** - ええと...ここには非常に有益な情報があります ;)
 - これは、全体の割り込みディスクリプター内のビット32～45です。
- **フラグ** - 楽しいことはどこにでもあります。
 - 割り込みディスクリプターのビット39～41。これは、ビットフラグ
 - 割り込みディスクリプタービット42～45。これがDPL（Descriptor Priveldge Level）です。

```
// ! 予約済み、0にすべき
uint8_treserved;

// ! ビットフラグ。
uint8_tfla
gs以上のフラグで設定します。

// ! IRアドレスのビット16～32
uint16_tbaseHi
```

```
;

#ifdef MSC_VER
#pragma pack (pop,
1) #endif
```

簡単ですね。この構造体が、割り込み記述子の構造と一致していることに注目してください。さて、割り込み記述子の説明ができたところで、IDTのインストールを見てみましょう。

idt.cpp - idtr

gdtrの構造と同じように、idtrの構造もあります。この構造が、idtrレジスタの構造とまったく同じであることに注目してください。

```
#ifndef _MSC_VER
#pragma pack (push,
1) #endif

// ! プロセッサのidtrレジスタの構造を記述しています struct idtr {

    // ! 割り込みディスクリプターテーブル(idt)のサイズ
    uint16_t limit;

    // ! idtのベースアドレス
    uint32_t base;
};

#ifdef _MSC_VER
#pragma pack (pop,
1) #endif

// ! 割り込みディスクリプターテーブル
static struct idt_descriptor _idt [I86_MAX_INTERRUPTS]です。

// ! CPUのidtrレジスタを定義するためのidtr構造体 static struct
    idtr_idtr;
```

なるほど...。IDTは、割り込みディスクリプターの配列に過ぎないことを覚えていますか？この場合、_idtrは参照のためのもので、プロセッサのIDTRレジスタに現在の情報を保存し、私たちが使えるようにします。基本的には、IDTと_idtrをセットアップし、IDTをインストールすればよいのです。難しいことはありません。)

idt_install() - 新しいIDTをインストールします。

これはIDTをIDTRにインストールするためのもので、それ以上でもそれ以下でもありません。これは、インラインアセンブリ言語（これはコンパイラに依存する）を共通のインターフェイスの後ろに抽象化し、コンパイラ間の移植性を助けるために使用されるヘルパーメソッドです。

```
// ! プロセッサにidtrをインストールする idtr
register static void idt_install () {
#ifdef _MSC_VER
    _asm lidt [_idtr]
#endif
}
```

i86_default_handler() - デフォルトの割り込みハンドラ

私たちのIDTインターフェイスは、独自の割り込み処理ルーチンをIDTに直接インストールする方法を提供します。割り込みが256個あるので、割り込みハンドラも256個あります。序盤ですべてを使うことはないでしょう。では、カーネルがまだ処理していない割り込みが発生した場合はどうなるのでしょうか？

これはそのためのものです。これは、IDTインターフェイスがインストールする基本的な未処理例外ハンドラです（これは後で見ることになります）。そして、システムを停止させます。

```
// ! 処理されないシステム割り込みをキャッチするデフォルトハンドラ
void i86_default_handler () {

#ifdef _DEBUG
    DebugClrScr(0x18);
    DebugGotoXY(0,0);
    DebugSetColor(0x1e)。
    DebugPrintf ("**** [i86 Hal] i86_default_handler: Unhandled Exception")と表示されます。
#endif

    for (;;)となっています。
}
```

割り込みから戻ると...

CやC++では、IRから戻るときに、自動的にスタックから値をポップしてRET命令を発行します。これではいけません。このため、私たちはIRET命令を使って戻る独自の方法を発行する必要があります。

geninterrupt() - 割り込みコールを生成する

これは少し厄介なことです。これは、インラインアセンブリ言語を共通のインターフェイスの後ろに抽象化して、より多くのコンパイラのために移植性を高めるために提供されているもう一つのヘルパーメソッドです。しかし、この方法では arbitrary interrupt call を生成するという課題も隠されています。

基本的には、INT OPCodeの2バイト目を変更すればよいのです。このコードは常に2バイト（1バイト目が0xCD、2バイト目が呼び出すべき割り込み番号）であることを知っていれば、非常に簡単な解決策となります。

パラメータとして関数ポインタを渡します。このルーチンは、ポインタが指す関数のアドレスを取得し、下位ビットと上位ビットをマスクして、**__idt**の構造体に格納します。**__idt [i]**はIDT内のディスクリプタ・オフセット（インタラプト番号）です。

```
int i86_install_ir (uint32_t i, uint16_t flags, uint16_t sel, I86_IRQ_HANDLER irq) {...
```

```

if (i>I86_MAX_INTERRUPTS)
    0を返す。

if (!irq)
    0を返す。

// ! 割り込みハンドラのベースアドレスの取得
uint64_t uiBase = (uint64_t)&(*irq)です。

IDTにベースアドレスを格納
_idt[i].baseLo      = uiBase & 0xffff;
_idt[i].baseHi      = (uiBase >> 16) & 0xffff;
_idt[i].reserved     = 0;
_idt[i].flags        = のフラグを立てます。
_idt[i].sel          = のセルです。

0を返す。

```

i86_idt_initialize () - IDTインターフェースの初期化

IDTの設定に使用されるビットフラグは**idt.h**で定義されており、コードをより読みやすく、修正しやすくするために提供されています。

```

idtの初期化
int i86_idt_initialize (uint16_t codeSel) {

    // ! プロセッサのidtrの設定
    _idtr.limit = sizeof (struct idt_descriptor) * I86_MAX_INTERRUPTS - 1;
    _idtr.base   = (uint32_t)&_idt[0];

    // ! idtを無効にする
    memset ((void*)&_idt[0], 0, sizeof (idt_descriptor) * I86_MAX_INTERRUPTS-1);

    // ! デフォルトのハンドラーを登録
    for (int i=0; i<I86_MAX_INTERRUPTS; i++)
        i86_install_ir (i, I86_IDT_DESC_PRESENT | I86_IDT_DESC_BIT32, codeSel,
                        (I86_IRQ_HANDLER)i86_default_handler);

    // ! 私たちのidtをイン
    ストールします
    idt_install () 。
}

```



```
}
0を返す。
}
```

デモの結末

今回のデモは、正直言ってちょっと複雑です。少なくとも、醜い必要性を排除することができました。INT命令を発行すると、デフォルトのハンドラが呼び出されることがわかると思います。独自の割り込みハンドラをインストールした場合は、エラーコードのあるものとなないものの両方を試してみてください。エラーコードのあるものとなないものの両方を試してみると、割り込みが発生するのがわかります。geninterrupt()やINT命令を呼び出すと、正しい割り込みハンドラ（または割り込みハンドラが定義されていない場合はデフォルトのハンドラ）が実行されるのがわかります。

このチュートリアルが複雑にならないように、ハードウェア割り込みはまだ取り扱わないことにしました。次のチュートリアルでは、カーネルのシステム・タイマーとして使用する**8253プログラマブル・インターバル・タイマー（PIT）**と、ハードウェア割り込みに必要な**8259Aプログラマブル・インタラプト・コントローラ**のコードを開発します。

デモをよく見て、すべてがどのように動くかを勉強してください。i86_install_ir()を使って自分の割り込みハンドラを登録してみてください。と割り込みを発生させます。そのために必要なのは

割り込みハンドラのパラメータは、割り込みハンドラが変わる可能性があるため、省くことにしました。そのため、パラメーターにアクセスするためには、ESPを介してアクセスする必要があります。後でパラメータを与えて楽にしようと思うかもしれませんが。

デモダウンロードはasmcpu(MSVC++)

結論

```
asm add esp, 12
// do whatever...
asm popad
```

今回は楽しいことがたくさんありましたね。このチュートリアルでは、多くのことを学びました。多くの重要なトピックを取り上げ、例外処理と割り込み処理を説明し、システムの割り込みを再び有効にできました。トリプルフォールトが発生するのもこれが最後かもしれません。わーい！

このチュートリアルは、正直言ってちょっと複雑です。OSプログラミングは楽しいでしょう？^_^

i86_install_ir (5, I86_IDT_DESC_PRESENT | I86_IDT_DESC_32BIT | I86_IDT_DESC_16BIT | I86_IDT_DESC_8BIT | I86_IDT_DESC_4BIT | I86_IDT_DESC_2BIT | I86_IDT_DESC_1BIT | I86_IDT_DESC_0BIT)に続いていきます。8259A PICチュートリアルと同様に、**8254プログラマブル・インターバル・タイマー（PIT）**（**8254 PIC HANDLER**）を使用してタイマーを処理する予定です。その後は、メモリ管理やプロセス管理を行う予定です。さらに、テキストベースの基本的なデバッグ用コンソールを開発して、状況を改善することも考えています。を少しカスタマイズして命令を生成します。インラインアセンブリーでももちろんgeninterrupt (5)が使えます。

次の機会まで。

マイク

BrokenThorn Entertainment社。現在、DoEとNeptuneOperating Systemを開発中です。質問やコメントはありますか？お気軽にお問い合わせください。

あなたも記事の改善に貢献したいと思いませんか？もしそうなら、ぜひ私に教えてください。



オペレーティングシステムの開発 - PIC、PIT、および例外処理

by Mike, 2008

このシリーズは、オペレーティングシステムの開発を一から実演し、教えることを目的としています。

注意：このチュートリアルでは、ハードウェア割り込みの処理を扱い、ソフトウェア割り込みの処理は扱いません。ソフトウェア割り込みをお探しの方は、[チュートリアル15](#)をご覧ください。このチュートリアルでは、ソフトウェア割り込み処理の知識が必要です。

はじめに

ようこそ！...え？もうチュートリアル16？

前回のチュートリアルでは、割り込み処理の世界に深く入り込みました。ハードウェア抽象化層の中のGDTとIDTのインターフェースを説明し、さらに実装しました。ソフトウェアによる割り込み処理に必要なものはほぼすべて網羅しました。.....しかし、待ってください。それはハードウェアインタラプトについて？

多くの重要なシステムデバイスが割り込みを使用しているため、ハードウェアデバイスによって引き起こされる割り込みを処理し、キャッチすることができる必要があります。良いニュースがあります。これはすでに実現されているのです。何が？**8259 Programmable Interrupt Controller (PIC)**です。次のセクションで詳しく見てみましょう。

ハードウェア割り込みが単独で動作するようになったとしても、システムタイマーの問題に直面することになります。システムタイマーが、私たちが設定した有効な割り込みハンドラを使用しない限り、ハードウェア割り込みを有効にした後、数ミリ秒後にトリプルフォールトになります。結局のところ、無効な割り込みハンドラを呼び出してしまうのです。そこで、**プログラマブルインターバルタイマ (PIT)** として知られるシステムタイマを再プログラムすることで、この小さな問題も解決します。

このチュートリアルには多くの内容が含まれて

- います。見ていきましょう。ハードウェアインタラプト
- 割り込みチェイニング
- Hal：プログラマブルインタラプトコントローラ Hal：プログラマブルインターバルタイマ
- ハードウェアアブストラクション
- インタラプトインプリメンテーションと私たちのHALのデザイン

ここでは、割り込み処理については触れていませんのでご注意ください。割り込み処理については、[チュートリアル15](#)を参照してください。これらのことを念頭に置いて、見てみましょう...

ハードウェアインタラプト

割り込みには、ソフトウェアで発生させるもの（INT、INT 3、BOUND、INTOなどの命令で使用）と、ハードウェアで発生させるものがあります。

ハードウェア割り込みは、PCにとって非常に重要です。他のハードウェアデバイスが、何かが起ころうとしていることをCPUに知らせることができます。例えば、キーボードのキーストロークや、内部タイマーの1クロック分の目盛りなどです。

これらの割り込みが発生したときに、どのようなIRQ（Interrupt Request）を生成するかをマッピングする必要があります。こうすることで、ハードウェアの変化を追跡することができます。

ここでは、これらのハードウェア割り込みについて説明します。

x86ハードウェアインタラプト		
8259A 入力端子	割り込み番号	説明
IRQ0	0x08	タイマー
IRQ1	0x09	キーボード
IRQ2	0x0A	8259Aスレーブコントローラ用カスケード
IRQ3	0x0B	シリアルポート2
IRQ4	0x0C	シリアルポート1
IRQ5	0x0D	ATシステム。パラレルポート2PS/2システム：予約
IRQ6	0x0E	ディスクドライブ
IRQ7	0x0F	パラレルポート1
IRQ8/IRQ0	0x70	CMOS リアルタイムクロック
IRQ9/IRQ1	0x71	CGAの垂直方向のリトレース
IRQ10/IRQ2	0x72	予約
IRQ11/IRQ3	0x73	予約
IRQ12/IRQ4	0x74	ATシステム：予約済みPS/2；補助的なデバイス
IRQ13/IRQ5	0x75	FPU
IRQ14/IRQ6	0x76	ハードディスク・コントローラー
IRQ15/IRQ7	0x77	予約

各デバイスについては、まだまだあまり気にする必要はありません。8259Aのピンについては、[8259PIC](#)のチュートリアルで詳しく説明しています。この表に記載されている割り込み番号は、これらのイベントが発生したときに実行されるデフォルトのDOS割り込み要求（IRQ）です。

ほとんどの場合、新しい割り込みテーブルを作り直す必要があります。そのため、ほとんどのオペレーティングシステムでは、PICが使用する割り込みをリマップして、IVT内の適切なIRQを呼び出すようにする必要があります。これは、リアルモードのIVTではBIOSが行ってくれます。このチュートリアルでは、後ほどこの方法を説明します。

www.brokenthorn.com/Resources/OSDev16.html

2021/11/15 13:08

オペレーティングシステム開発シリーズ

待ってください。 このPICというのは何ですか？ハードウェアデバイスに信号を送ることができるこれらのハードウェアデバイスはすべて、間接的に**8259A Programmable**に接続されています。

割り込みコントローラ (PIC)。これは特別で非常に重要なマイクロコントローラーで、マイクロプロセッサがハードウェア割り込みをかける必要があるときに信号を送るために使用されます。

このマイクロコントローラーのプログラミングは、このチュートリアルの後に行います。このマイコンはかなり複雑なので、別のチュートリアルを用意しました。[こちらをご覧ください。](#)

割り込みチェイニング

IDT (Interrupt Descriptor Table) の中に独自の割り込みハンドラを簡単にインストールできるようになります。割り込みハンドラは、ソフトウェア割り込みだけでなく、ハードウェアデバイスによって引き起こされる割り込みにも対応できるように作成します。**覚えておいてください。ハードウェアデバイスは Programmable Interrupt Controller に信号を送り、プロセッサにハードウェア割り込みのトリガを要求します。** PIC はプロセッサに、どの **割り込み要求 (IRQ)** を呼び出すかを、**IDT (Interrupt Descriptor Table)** の中で知らせます。

しかし、待ってください…。PIC は、私たちの IDT の中でどの **IRQ** を呼び出すべきかをどうやって知るのでしょうか？それを伝えます。

そのため、どの割り込みを使うかを PIC に知らせるために、PIC を再プログラムする必要があります。

さて、ソフトウェアやハードウェアの割り込みを処理する割り込みハンドラができたとして。これでどうでしょう？私たちの立場からすると、これはどうでしょうか？確かに、異なるデバイス用のハンドラを簡単にインストールできますが、複数のデバイスが同じ割り込みを必要とする場合はどうでしょうか？1つのソフトウェア割り込みに複数の機能が必要な場合は？そこで、**インタラプト・チェイニング**の出番です。

割り込みチェーンとは、同じ割り込み番号を共有するすべての割り込みハンドラを復元して呼び出すための技術です。これは、以前の **割り込みルーチン (IR)** を関数ポインタに保存することで行われます。その後、新しいハンドラをインストールし、新しい **IR** が呼び出されるたびに前の割り込みハンドラを呼び出します。

ここではその一例をご紹介します。

ご覧のように、割り込みの連鎖はとても簡単です。 **setvect()** は新しい割り込みベクターを設定し、 **getvect()** は割り込みベクターを返します。これらの割り込みベクターは、 **IVT (Interrupt Vector Table)** または **IDT (Interrupt Descriptor Table)** のいずれかに格納できます。待って、何？そうです、私たちです。)

準備する - 割り込み処理の実装

割り込みと割り込み処理については、かなりの部分をカバーしています。テキストだけでは限界があります。ハードウェアの割り込み処理がどのように機能するかについても少し調べましたが、ここまで来るとは十分ではありませんでした。

プログラマブル・インターラプト・コントローラ のプログラム方法を学ぶまでは、ハードウェア割り込み処理を実装することはできません。また、タイミングの問題を解決するまでは、ハードウェア割り込みを有効にすることはできません (BIOS のおかげで、 **Programmable Interval Timer** がまだ **IRQ8** に接続されていることを覚えていますか？つまり、 **BIOS が割り込みを再び有効にする** とすぐに、次のタイマーチェックがダブルフォールトになってしまうのです)。このため、 **Programmable Interval Timer** を再び有効にする方法も習得しなければなりません。

読者の皆さん、ここからが複雑なのです。 **ハードウェア・プログラミングの世界へ、ようこそ！**

しかし、良いニュースもあります…。これらのマイクロコントローラーは、どれもそれほど複雑ではありません。しかし、メインのシリーズが複雑になりすぎないように、これらの **マイクロコントローラーに特化した2つのチュートリアル** を書くことにしました。この先のデモやコードを理解するための必読書です。

このため、読者の皆様には、次の割り込みハンドラを読んでから先に進むことをお勧めします。 **8259 プログラマ**

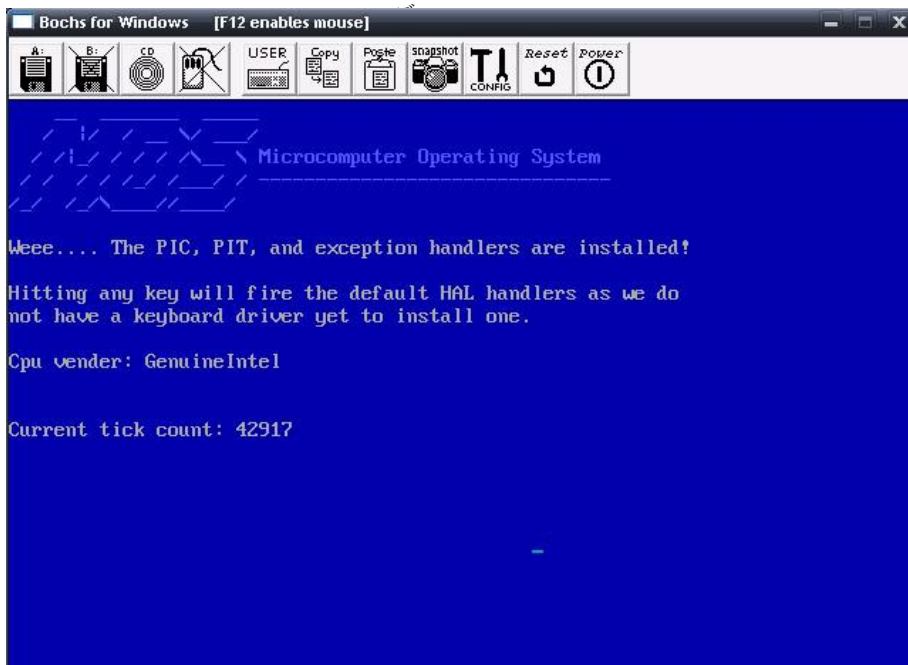
- **フル・インタラプト・コントローラ** ***(prevhandler) を呼び出す ()**。
- **8253 プログラマブル・インターナバル・ダイマ**

これらのチュートリアルをすべてを理解していなくても心配しないでください。この先のセクションでは、上記のチュートリアルをすべてを実装していきます。:) また、ここではまだすべてを説明していますので、あなたが暗闇に導くことはありません。

また、上記のチュートリアルを参考にしてくださいとよいでしょう。

それで...？何を待っているの？チュートリアルに飛び込んでみましょうそして、終わったらここに戻ってきてください。心配しないで、私はただのテキストだから…。心配しないでください、私はただのテキストです...あなたが戻ってきたときにまだここにいます。もちろん、削除されていなければですが！)

デモ



新デモの動作

デモダウンロード(MSVC++)

正直に言うと、このデモは少し複雑です。しかし、ほとんどのコードは見覚えがあるはずなので、それほど難しいものではありません。

このデモでは、**ハードウェア抽象化層 (HAL)** を使用して、独自の例外ハンドラをインストールしています。また、HALを初期化し、プロセッサテーブル、PIC、PITを初期化しています。これにより、ハードウェア例外を有効にすることができます（ついに！）。**このデモでは、（PITの割り込みハンドラによって更新された）現在の目盛りを画面に表示します。**その他の割り込みが発生した場合は、IDTのデフォルトの割り込みハンドラで処理されます。これはIDTのチュートリアルで設定しましたね。

キーボードはIRQ9を生成しており、まだそのための割り込みを定義していないので、キーが押されるとIDTのデフォルトハンドラが実行されます。

さて... 今回のデモは、他のデモよりも少しクールに見えるようにしてみました。同時に、これ以上複雑なものを増やしたくはありませんでした。PICとPITマイクロコントローラのインターフェースを作ることを見てきたので、デモのコード自体を少し見てみましょうか...

ハードウェアの抽象化

最初に見ていくのは、ハードウェア抽象化レイヤーが提供するインターフェースです。これは、**include/hal.h**と**hal/hal.cpp**を見ればわかります。ルーチンのほとんどは非常にシンプルで、これまで開発してきた（そしてこれから開発する）他のインターフェース（GDT、IDT、CPU、PIC、PITなど）を単純に使用しているだけなので、深くは説明しません。その代わりに、インターフェースそのものを見てみたいと思います。これは、カーネルとデバイスドライバが使用するインターフェースになるので、なぜそうしないのか？

新しい hal.h

このあたりから、ハードウェアの抽象化がいかに有効であるかが見えてきます。私は、16ビットのDOSをプログラミングするのと同じくらい簡単に使える「DOS」のようなインターフェースを提供したいと考えました。そのために、様々な目的に使用できるルーチンの簡単なリストを用意しました。これらのルーチンを見ると、使用されているハードウェアデバイスやテーブルへの言及が全くないことがわかります。これこそが、ハードウェアの抽象化です。アーキテクチャを抽象化するのではなく、使用しているハードウェアを抽象化するのです。

後に使用するコードの多くは、HAL内のルーチンを使用してタスクを実行しています。そのため、今回はハードウェア抽象化レイヤーと、それが提供するルーチンを見ていただきたいと思います。

```

エクスタ int          _cdecl hal_initialize ()です。
_cdecl hal_shutdown ()です。
エクスタ int          _cdecl enable ()です。
_cdecl disable ()です。
エクスタ ボイド       _cdecl geninterrupt (int n)。
_cdecl inportb (unsigned short id);
エクスタ unsigned char _cdecl outportb (unsigned short id, unsigned char value);
_cdecl setvect (int intno, void (_cdecl far &vect) ( ))
_cdecl getvect (int intno) ( );
エクスタ void (_cdecl) far * _cdecl interruptmask (uint8_t intno, bool enable);
_cdecl interruptdone (unsigned int intno);
エクスタ inline void _cdecl sound (符号なしの周波数)。
_cdecl get_cpu_vender ()です。
エクスタ const char* _cdecl get_tick_count ()です。

```

16bit DOSをプログラムしたことがある人は、今すぐにも家にいるような気分になれるはずです。:)

プログラマブルインタラプトコントローラ

8259:マイクロコントローラ

8259マイクロコントローラファミリは、**PIC(Programmable Interrupt Controller)集積回路(IC)**のセットです。ハードウェアコントローラは、ハードウェア割り込みが要求されると、PICに間接的に接続されます。このため、ハードウェア割り込みを処理するためには、このマイクロコントローラのプログラム方法を理解する必要があります。

ここではまだすべてを説明しますが、8259は複雑なマイクロコントローラです。そのため、このコントローラだけをカバーするために、完全なチュートリアルを用意しました。そのため、このセクションを最大限に活用するには、以下のチュートリアルを参照してください。

8259Aプログラマブルインタラプトコントローラチュートリアル

注意：ここでは、PICやハードウェアの割り込み処理に関するすべてをカバーするわけではありません。これについては上記のチュートリアルをご覧ください。

8259:概要

PIC (Programmable Interrupt Controller) は、割り込みラインを介してデバイスとプロセッサを接続するためのマイクロコントローラーです。これにより、デバイスは、システムソフトウェアやエグゼクティブの注意を必要とするときはいつでもプロセッサに信号を送ることができます。これが**IRQ (Interrupt Request)** です。

PICは、すべてのハードウェア割り込み要求を制御します。これにより、さまざまなハードウェアデバイスが注意を必要とするたびに、そのデバイスから信号を受け取ることができます。**フロッピーディスクコントローラー (FDC)** などのデバイスが注意を必要とするときは、PICに割り当てられたIRQを起動するように指示します。ここで、PICはプロセッサに信号を送り、呼び出すべき割り込み番号を伝えます。その後、プロセッサはIDTにオフセットし、リング0で割り込みハンドラを実行します。すべての割り込みハンドラを定義したので、いよいよ制御を開始します。

この一番の利点は、PICのおかげですべてが**自動で行われる**ことです。デバイスがPICに信号を送ると、私たちの割り込みハンドラが自動的に実行されます。また、プロセッサはリング0への**タスクスイッチ**を実行するので、リクエストを処理するために常にカーネルランドにたどり着きます。かっこいいでしょう？

PIC自体は複雑なマイクロコントローラーです。ここでは、すべてを詳しく説明しようと思いますが、このチュートリアルを最大限に活用するために、読者の皆様には上記のPICチュートリアルを読んでいただくことをお勧めします。

以上のことを念頭に置いて、インターフェースに飛び込んでみましょう。これらのコードはすべて、このチュートリアルの最後にあるデモで見ることができます。

操作コマンド

オペレーションコマンドは、ビットパターンで構成された特殊なコマンドです。このビットパターンは、マイコンにコマンドを記述するために設定する必要があります。操作コマンドには基本的に2種類あります。**ICW (Initialization Command Words)** と **OCW (Operation Command Words)** です。

ICWは、デバイスの初期化時にのみ使用される操作コマンドです。OCWは、デバイスの初期化後にデバイスを制御するために使用されます。

pic.h:インターフェース

このファイルは、システムの残りの部分の全体的なミニドライバのインターフェースを提供します。これは、PICを制御・管理するためのインターフェースです。私は「ミニドライバ」を、単体のソフトウェアではなく、ソフトウェアの一部に組み込まれたドライバと定義しています。

pic.h:デバイスの接続

PICチュートリアルでは、ハードウェア割り込みについて深く掘り下げてみました。ハードウェアデバイスがシステムソフトウェアやエグゼクティブの注意を必要とするときに、どのようにPICに信号を送るかを見てきました。これを実現するために、各デバイスはPIC上の**IR (Interrupt Request)** ラインに間接的に接続されています。このラインは、デバイスが使用する**割り込み要求 (IRQ)** を表すだけでなく、そのプライオリティー・レベルも表しています (IRQ番号が低いほど、プライオリティーが高くなります)。

個々のデバイスとそのIRQを扱う際に役立つように、そのデバイスが使用するIRQを抽象化したいと考えています。これは、移植性を高めるだけでなく、定数の後ろに表示されるので読みやすくなります。覚えておってください。魔法の数字は悪いものです。

```
#!/この以下のデバイスは、PIC 1を使って割り込みとです。
#define i86_pic_irq_timer 0
#define i86_pic_irq_keyboard 1
#define i86_pic_irq_serial2 3
#define i86_pic_irq_serial1 4
#define i86_pic_irq_parallel2 5
#define i86_pic_irq_diskette 6
#define i86_pic_irq_parallel1 7

#!/この以下のデバイスは、PIC 2を使って割り込みとです。
#define i86_pic_irq_cmostimer 0
#define i86_pic_irq_cgaretrace 1
#define i86_pic_irq_auxiliary 4
#define i86_pic_irq_fpu 5
#define i86_pic_irq_hdc 6
```

上記の定数は、使用しているすべてのデバイス (IRQライン/番号も一緒に) をリストアップしています。**1つのPICには8本のIRラインしかないため、1つのPICには8つのIRQしか設定できません。**一般的なx86アーキテクチャでは、プライマリとセカンダリの2つしかありませんが、**PICはセカンダリのPICとカスケード接続できます**。

今、私たちにとって最も重要な2つのデバイスは、タイマー(I86_PIC_IRQ_TIMER)とキーボード(I86_PIC_IRQ_KEYBOARD)です。このチュートリアルでは、I86_PIC_IRQ_TIMERを使用しますので、すべてがどのように連動するかがわかります。

pic:8259のコマンド

PICの設定は非常に複雑です。PICの設定は、一連の**コマンド・ワード**によって行われます。コマンド・ワードとは、初期化や動作に使用される様々な状態を含むビット・パターンです。少し複雑に見えるかもしれませんが、それほど難しいことはありません。まず、PICの制御に使われる**OCW (Operation Command Word)** を見てみましょう。初期化コマンドについてはもう少し後に説明します。

pic:操作コマンドワード1

これは、**IMR (Interrupt Mask Register)** の値を表しています。特別なフォーマットを持たないため、ハードウェアの割り込みを有効/無効にするためのインプリメンテーションファイルで直接処理されます。サイズは1バイトです。正しいビットを設定することで、割り込み要求ラインの有効化と無効化 (「マスクとアンマスク」) を行います。1つのPICには8つのIRQしかないことを覚えていませんか？つまり、IMRのビット0がIRQ 0、ビット1がIRQ 1、ビット2がIRQ 2、という具合です。

この後、**Interrupt Maskレジスタ**を見てみましょう。

pic:操作コマンドワード2

これは、PICを制御するための主要なコントロールワードです。それでは見てみましょう。

オペレーティングシステム開発シリー		
オペレーションズ コマンド・ワード (OCW) 2		
ビット数	価値	説明
0-2	L0/L1/L2	コントローラが反応すべきインタラプトレベル

3-4	0	予約済み、0でなければならない
5	EOI	EOI (End of Interrupt) リクエスト
6	SL	セクション
7	R	回転オプション

それじゃ！」と。

OCW 2のフォーマットはとても簡単です。最初の3ビットは現在のインタラプトレベルです。3-4ビットは予約です（0でなければなりません）。第5ビットは**EOI (End of Interrupt)**を表します。第6ビットは**選択**ビットです。ビット7は、**回転**コマンドを提供します。

各コマンドは個々のビットで選択されているので、これらのコマンドを**ビットワイズOR**してOCW2を生成します。

```
#!/コマンドワード2のビットマスク。コマンド送信時に使用
#define i86_pic_ocw2_mask_l1 1 //00000001 //レベル1の割り込みレベル
#define i86_pic_ocw2_mask_l2 2 //00000010 //レベル2の割り込みレベル
#define i86_pic_ocw2_mask_l3 4 //00000100 //レベル3の割り込みレベル
#define i86_pic_ocw2_mask_eoi 0x20 //00100000 //インタラプトコマンドの終了
#define i86_pic_ocw2_mask_sl 0x40 //01000000 //選択コマンド
#define i86_pic_ocw2_mask_rotate 0x80 //10000000 //ローテーションコマンド
```

これは、私たちにとって重要な命令語です。これは私たちにとって重要なコマンドワードです。すべての割り込みハンドラからこのコマンドワードを送信する必要があります。

PICが実行されたときに割り込みをマスクオフしたことを覚えていますか？これは、プロセッサが**PIC**を確認するまで、その**IR**ラインの割り込み要求はそれ以上実行できないということです。これは、正しい**PIC**に**End of Interrupt**コマンドワードを送信することで行われます。これは、コマンドワードのEOIビットを**マスクオフ**することで行うことができます。これは、**I86_PIC_OCW2_MASK_EOI**が使用されるものです。

この後、インターフェイスには、PICにコマンドを送信するための**i86_pic_send_command**ルーチンがあることがわかります。このルーチンを使ってEOIコマンドを送信する例を見て、どのように動作するかを確認しましょう。

```
i86_pic_send_command (I86_PIC_OCW2_MASK_EOI, picNumber);
```

上記のコードは、**picNumber**に指定された写真にEOIコマンドを送信します。以

上、OCW2の紹介でした。次の作品に期待しましょう。

pic:操作コマンドワード3

*このセクションに追加することを計画しています。

```
#!/Command Word 3 ビットのマスク。コマンド送信時に使用
#define i86_pic_ocw3_mask_ris 1 //00000001
#define i86_pic_ocw3_mask_rir 2 //00000010
#define i86_pic_ocw3_mask_mode 4 //00000100
#define i86_pic_ocw3_mask_smm 0x20 //00100000
#define i86_pic_ocw3_mask_esmm 0x40 //01000000
#define i86_pic_ocw3_mask_d7 0x80 //10000000
```

pic.cpp:インプリメンテーション

さて...ここまではすべて簡単でしたよね？ここまでは簡単でしたが、「課題はどこにあるの？」とお思いでしょう。pic.cpp

は、PICインターフェースの実装を行います。まず最初に見なければならないのは、レジスタです。**pic.cpp:レジスタ**

定数

ここでは、PICのポート位置を抽象化するための定数を定義しています。すべてのレジスタ名に定数を定義していることに注目してください。同じポートアドレスを共有しています。その理由は、完全性のためです。同じポートの位置を共有していても、異なるレジスタであることに変わりはありません。

```
#!/PIC 1 レジスタポートアドレス
#define I86_PIC1_REG_COMMAND 0x20 // コマンドレジスタ
#define i86_pic1_reg_status 0x20 ステータスレジスタ
#define i86_pic1_reg_data 0x21 // データレジスタ
#define i86_pic1_reg_imr 0x21 // 割り込みマスクレジスタ (imr)

#!/PIC 2 レジスタポートアドレス
#define I86_PIC2_REG_COMMAND 0xA0 // ^ 上記のレジスタ名を参照
#define i86_pic2_reg_status 0xA0
#define i86_pic2_reg_data 0xA1
#define i86_pic2_reg_imr 0xA1
```

難しいことはありません。**コマンドレジスタ**にコマンドを送り、**データレジスタ**からデータを読み出します。データレジスタから書き込む場合は、割り込み要求を手動でマスクしたり、マスクを解除したりするために使用できる**割り込みマスクレジスタ (IMR)**にアクセスしていることになります。このようにして、割り込み要求を有効または無効にすることができます。

アクセスしているレジスタは、書き込みか読み出ししかによって異なります。ポート0x20に**書き込む**場合は、コマンドレジスタにアクセスしています。ポート0x20に**書き込む**場合は、コマンド・レジスタにアクセスし、ポート0x20から**読み出す**場合は、ステータス・レジスタにアクセスしています。

最後に、これは暗黙の了解であるため、インターフェースではなく暗黙の了解 (pic.cpp) の一部となります。次

に、初期化時に使用される定数を見てみましょう。

pic.cpp:初期化制御ワード1



初期化コントロールワード (ICW) 1		
ビット数	価値	説明
0	IC4	セット (1) の場合、PICは初期化中にIC4を受信することを期待します。
1	SNGL	セット (1) の場合、システム内のPICは1つだけです。クリアした場合、PICはスレーブPICとカスケード接続され、ICW3をコントローラに送る必要があります。
2	ADI	設定(1)されている場合、CALLアドレス間隔は4、それ以外は8となります。これはx86では通常無視され、デフォルトでは0に設定されています。
3	LTIM	(1)に設定されている場合、レベルトリガモードで動作します。未設定(0)の場合、エッジトリガモードで動作します。
4	1	初期化ビット。PICを初期化する場合は1を設定
5	0	MCS-80/85：インタラプト・ベクタ・アドレス x86アーキテクチャ。0でなければならない
6	0	MCS-80/85：インタラプト・ベクタ・アドレス x86アーキテクチャ。0でなければならない
7	0	MCS-80/85：インタラプト・ベクタ・アドレス x86アーキテクチャ。0でなければならない

ご覧のように、ここにはたくさんのことが起こっています。これらのうちのいくつかは以前に見たことがあります。これらのビットのほとんどはx86プラットフォームでは使用されていないので、これは思ったほど難しくありません。

各コマンドワードには2種類の定数があります。1つ目のタイプは**ビットマスク**で、データが表すビットをマスクするために使用されます。2つ目のタイプの定数は、**コマンドコントロールビット**で、マスクと一緒に使用され、正しい値に設定されます。

もっと詳しく見てみましょう。ICWの1ビットマスクを見てみましょう。上の表に示されたフォーマットに従っていることに注目してください。最後の3ビットはx86アーキテクチャでは常に0であるため、ここでは何も定義しません。

```
#!/ 初期化制御ワード1ビットのマスク
#define I86_PIC_ICW1_MASK_IC4      0x1      //00000001      // ICW 4ビットを期待する
#define I86_PIC_ICW1_MASK_SNGL     0x2      //00000010      シングルまたはカスケード
#define I86_PIC_ICW1_MASK_ADI      0x4      //00000100      // コール・アドレス・インター
                                           バル
#define I86_PIC_ICW1_MASK_LTIM     0x8      //00001000      操作モード
#define I86_PIC_ICW1_MASK_INIT     0x10     //00010000      初期化コマンド
```

さて、上記のビットマスクを使ってICWのビットを1に設定することは簡単にできますが、その意味をどうやって知ることができるでしょうか？つまり、設定したいビットをマスクしたときに、その設定値の意味をどうやって知ることができるのでしょうか？ここで、**コマンドコントロールビット**の出番です。

これらは、上記のマスクされたビットを設定するために使用できる定数値を含んでいます。これにより、読みやすさと拡張性が大幅に向上します。ICW 1の

コマンドコントロールビットを示します。見てみましょう...

```
#define i86_pic_icw1_ic4_expect    1          //1          //I86_PIC_ICW1_MASK_IC4の設定時に使用します。
#define i86_pic_icw1_ic4_no        0          //0
#define i86_pic_icw1_sngl_yes      2          //10         //I86_PIC_ICW1_MASK_SNGLの設定時に使用します。
#define i86_pic_icw1_sngl_no       0          //00
#define i86_pic_icw1_adi_callinterval4 4        //100        //I86_PIC_ICW1_MASK_ADIの設定時に使用します。
#define i86_pic_icw1_adi_callinterval8 0        //000
#define i86_pic_icw1_ltim_leveltriggered 8       //1000       //I86_PIC_ICW1_MASK_LTIMの設定時に使用します。
#define i86_pic_icw1_ltim_edgetriggered 0        //0000
#define i86_pic_icw1_init_yes      0x10       //10000      //I86_PIC_ICW1_MASK_INITの設定時に使用します。
#define i86_pic_icw1_init_no       0          //00000
```

難しいことはありません。この命名規則により、何をどこで使用するかが簡単にわかります。例えば、**I86_PIC_ICW1_SNGL_YES**は、次のように使われます。**I86_PIC_ICW1_MASK_SNGL**、**I86_PIC_ICW1_LTIM_EDGETRIGGERED**は、**I86_PIC_ICW1_MASK_LTIM**と一緒に使用されます。

ここでは、それらがどのように連携するかの例を示します。PICを初期化する際に、初期化を有効にしてICW4を送信する必要があります。 そのためには、ICW1を次のように設定するだけです。

そうなんだ！？それは、とてもうまく機能し、調和していることに注目してください。このコードは、特定のビット（または一連のビット）を既知の値に設定するために、さまざまな場所で使用されたPICのマスクを見ただけで、何をCでしているのが分かるのが良いところです。(初期化を開始し、ICW 4を期待する)。) かつ、このコードは、I86_PIC_ICW1_MASK_IC4、I86_PIC_ICW1_MASK_SNGL、I86_PIC_ICW1_MASK_ADI、I86_PIC_ICW1_MASK_LTIM、I86_PIC_ICW1_MASK_INITの5つのマスクを、I86_PIC_ICW1_MASK_INITにマスキングの際に、必要に応じてこの方法を使用します。

初期化制御ワード2

このコントロールワードは、PICが使用するIVTのベースアドレスのマッピングに使用されます。

初期化コントロールワード (ICW) 2		
ビット数	価値	説明
0-2	A8/A9/A10	MCS-80/85モード時のIVT用アドレスビットA8-A10。
3-7	a11(t3)/a12(t4)/a13(t5)/a14(t6)/a15(t7)	MCS-80/85モード時のIVT用アドレスビットA11~A15。80x86モードでは、 割込みベクターアドレスを指定します 。x86モードでは0に設定してもよい。

初期化の際には、ICW2をPICに送り、使用するIRQのベースアドレスを伝える必要があります。ICW1をPICに送った場合（初期化ビットが設定されている場合）、次にICW2を送る必要があります。そうしないと、定義されていない結果になることがあります。誤った割り込みハンドラが実行される可能性が高くなります。

このコマンドは複雑なフォーマットを持たないため、**pic.cpp**の内部で直接処理され、定数はありません。

初期化制御ワード3

このコマンドワードは、PICコントローラがどのようにカスケード接続されているかを知らせるために使用されます。複数のPICをカスケード接続するには、PICのIRラインの一つを互いに接続する必要があります。それがどのラインなのかを知らせるために、このコマンドワードを使います。

初期化コントロールワード (ICW) 3		
ビット数	価値	説明
0-7	S0-S7	スレーブPICに接続するIRQ (Interrupt Request) を指定します。

このコマンドは複雑なフォーマットを持たないため、pic.cppの内部で直接処理され、定数はありません。

初期化コントロールワード 4

イエーイ!これは、最終的な初期化コントロールワードです。これは、すべての動作を制御します。

初期化コントロールワード (ICW) 4		
ビット数	価値	説明
0	UPM	セット(1)の場合、80x86モードである。MCS-80/86モードの場合はクリアされます。
1	AEOI	セットされていると、最後のインタラプトアクリッジパルスで、コントローラは自動的にEOI (End of Interrupt) 動作を行います。
2	M/S	BUFが設定されている場合のみ使用。1に設定されている場合、バッファーマスターが選択されます。バッファースレーブの場合はクリアされます。
3	BUF	設定されている場合、コントローラはバッファードモードで動作します。
4	SFNM	特別なフルネストモード。カスケード接続されたコントローラが多数存在するシステムで使用されます。
5-7	0	予約済み、0でなければならない

これはかなり複雑なコマンドワードですが、悪くないと思います。では、定義したビットマスクを見てみましょう。上のようなフォーマットになっていることに注目してください。

```
#!/初期化制御ワード 4ビットマスク
#define I86_PIC_ICW4_MASK_UPM          0x1      //00000001      モード
#define I86_PIC_ICW4_MASK_AEOI        0x2      //00000010      // 自動EOI
#define I86_PIC_ICW4_MASK_MS          0x4      //00000100      // バッファタイプの選択
#define I86_PIC_ICW4_MASK_BUF         0x8      //00001000      // バッファードモード
#define I86_PIC_ICW4_MASK_SFNM        0x10     //00010000      // 特別なフルネストモード
```

ICW 1と同様に、プロパティを設定するためにビットマスクと組み合わせる制御ビットのセットがあります。その内容は...

```
#define i86_pic_icw4_upm_86mode      1      //1      //I86_PIC_ICW4_MASK_UPMの設定時に使用します
#define i86_pic_icw4_upm_mcsmode     0      //0      。
#define i86_pic_icw4_aeoi_autoeoi    2      //10     //I86_PIC_ICW4_MASK_AEOIの設定時に使用します
#define i86_pic_icw4_aeoi_noautoeoi  0      //00     。
#define i86_pic_icw4_ms_buffermaster  4      //100     //I86_PIC_ICW4_MASK_MSの設定時に使用します。
#define i86_pic_icw4_ms_bufferslave   0      //000     。
#define i86_pic_icw4_buf_modeyes      8      //1000    //I86_PIC_ICW4_MASK_BUFの設定時に使用します
#define i86_pic_icw4_buf_modeno       0      //0000    。
#define i86_pic_icw4_sfnm_nestedmode  0x10   //10000   //I86_PIC_ICW4_MASK_SFNMの設定時に使用します
#define i86_pic_icw4_sfnm_notnested   0      //00000   。
```

これはシンプルなスナフキンですね。^_^ 上記の制御ビットとビットマスクを組み合わせ、制御ワードを構築していきます。名前の付け方によって、どのビットマスクと一緒に使われているかを簡単に識別することができます。

これで、実装に使われる定数は終わりだと思います。それでは早速、関数をご紹介します。

i86_pic_send_command ()です。PICにコマンドを送信する

このルーチンは、PICのコマンドレジスタにコマンドバイトを送信します。picNumは、アクセスするPICを表すゼロベースのインデックスです。x86では、これは0または1でなければなりません。正しいコマンドレジスタを取得するために、どのPICを使用しているかをテストすることに注意してください。

これはインターフェイスの一部ですが、インターフェイスの外ではあまり使用しないでください。これは、必要に応じてPICを手動で送信・制御するための方法を提供するものです。これは、EOIコマンドを送信するための割り込みハンドラが必要となります。

```
i86_pic_send_data ()およびi86_pic_read_data ()です。PICとの間でデータバイトを送受信する
これらのルーチンは上のルーチンと非常に似ていますが、picNumのPICに応じてPICのデータレジスタに書き込んだり読み込んだりします。これらのルーチンが両方ともインラインであることに注目してください。これらのルーチンは小さいので、関数呼び出しを削除したいと思います。

uint8_t reg = (picNum==1) ?I86_PIC2_REG_COMMAND : I86_PIC1_REG_COMMAND;
inline void i86_pic_send_data (uint8_t data, uint8_t picNum) {...
}

if (picNum > 1)
    を返すことができます。

uint8_t reg = (picNum==1) ?I86_PIC2_REG_DATA : I86_PIC1_REG_DATA;
outportb (reg, data);
}

インライン uint8_t i86_pic_read_data (uint8_t
picNum) { if (picNum > 1)
    0を返す。

uint8_t reg = (picNum==1) ?I86_PIC2_REG_DATA : I86_PIC1_REG_DATA;
return inportb (reg);
}
```

i86_pic_initialize ():PICを初期化する

これはPICインターフェースの最終ルーチンです。これは、上記のすべてのルーチンと、初期化制御ワード用に定義された定数を使用して、動作のために両方のPICを初期化します。

このルーチンはあまり複雑ではありません。というか、見た目ほど複雑ではありません;)このルーチンがすることは、PICに初期化コマンドを送ることだけです。これは、コマンドワードの**I86_PIC_ICW1_INIT_YES**ビットをセットすることで行います。また、**I86_PIC_ICW1_IC4_EXPECT**ビットを設定します。これはコントローラがICW 4を送ることを期待していることを保証するものです。定数が読みやすさを向上させていることに気がつきましたか？

ICWは...そう...icwに格納されています。**i86_pic_send_command()**ルーチンを使って、両方のPICにコマンドを送ります。

ICW 1が送られた後、ICW 2を送って初期化を開始します。ICW 2にはベースとなる割り込み番号が含まれていることを覚えていますか？これは次のように渡されます。**base0**と**base1**のパラメータです。

ICW 3は、マスターとセカンダリのPICコントローラーを接続するためのものです。

最後にICW4です。**I86_PIC_ICW4_UPM_86MODE**ビットを設定してx86モードを設定します。このルーチンを、[PICチュートリアル](#)にある例と比較してみてください。

```

//!picの初期化
void i86_pic_initialize (uint8_t base0, uint8_t base1)
{
    ...
    uint8_t      icw      = 0;

    //!PICの初期化開始

    icw = (icw & ~I86_PIC_ICW1_MASK_INIT) | I86_PIC_ICW1_INIT_YES;
    icw = (icw & ~I86_PIC_ICW1_MASK_IC4) |
    I86_PIC_ICW1_IC4_EXPECT;

    i86_pic_send_command (icw, 0) 。
    i86_pic_send_command (icw, 1) 。

    //!初期化制御ワード2を送信します。これはirqのi86_pic_send_dataのベースアドレス (base0,
    0)です。
    i86_pic_send_data (base1, 1) 。

    //!初期化制御ワード3を送信します。これでマスターとスレーブの接続は完了です。
    //!マスターPICのICW3は、セカンダリPICに接続するIRをバイナリ形式で表しています
    //!セカンダリPICのICW3はマスターPICに接続するIRを10進数で表したもの

    i86_pic_send_data (0x04, 0);
    i86_pic_send_data (0x02, 1)です。

    //!初期化制御ワードの送信 4.i86モードを有効にする

    icw = (icw & ~I86_PIC_ICW4_MASK_UPM) | I86_PIC_ICW4_UPM_86MODE;

    i86_pic_send_data (icw, 0) .
    i86_pic_send_data (icw, 1) .
}

```

Whew, これでPICの大掛かりな作業は終わりだね。あとはPITの再プログラムだけだ。心配しなくても、PICほど複雑ではありません。ちょっと見てみましょうか。

プログラム可能なインターバルタイマー

なるほど・・・。PICの準備ができたので、ハードウェア割り込みを有効にすることができるようになりましたよね？うん、ちょっとね。ここまでは問題ないのですが、まだPIT用の割り込みハンドラがインストールされていません。では、次のタイマーの目盛りで何が起こるのでしょうか？ ...私が何を言いたいのか、お分かりになると思います :)

プログラマブル・インターバル・タイマー(PIT)は、プログラムされたカウントに達すると、割り込みを発生させるカウンタです。**8253**および**8254**マイクロコントローラは、i86アーキテクチャに対応したPITで、i86対応システムのタイマーとして使用されます。

x86アーキテクチャでは、**PITはシステムタイマーとして機能し、PICのIROラインに接続されています**。これにより、PITはタイマーの目盛りごとに**IRQ 0**を発射することができます。このため、このマイクロコントローラーを使用する前に、再プログラムする必要があります。

PITはプログラムが複雑なマイクロコントローラーです。そのため、別のチュートリアルを用意しました。このチュートリアルでは、**PITのすべての機能を紹介します**つもりはありません。

PITについては、以下のチュートリアルを参照してください（参照してください）。

[8253プログラマブル・インターバル・タイマーチュートリアル](#)

pit.h:インターフェース

PITの良いところは、プログラムがそれほど複雑ではないことです。それほど多くのコマンドを含んでいませんし、かといってそれほど多くのコマンドを必要としません。ハードウェアのタイミングやリクエストに使われる、小さくてもパワフルなチップです。

操作コマンドワード

PITには、カウンタを初期化するための**OCW (Operation Command Word)** が1つだけ含まれています。このワードは、カウンタのカウントモード、オペレーションモードを設定し、初期カウント値を設定することができます。

このコマンド・ワードは少し複雑です。ここでは完全なコマンドワードをご紹介します。

- **ビット0: (BCP)** バイナリカウンタ
 - **0**: バイナリ
 - **1**: BCD (バイナリーコード化された10進法)
- **ビット1~3: (M0, M1, M2)** 動作モード。それぞれの説明は上記のセクションを参照してください。
 - **000**: モード0: インタラプトまたはターミナルカウント
 - **001**: モード1: プログラマブルワンショット
 - **010**: モード2: レートジェネレータ
 - **011**: モード3: 矩形波ジェネレータ
 - **100**: モード4: ソフトウェア・トリガ・ストロブ

2021/11/15 13:08

- **101** : モード5 : ハードウェア・トリガ・ストローブ
- **110**: 未定義、使用しないでください

オペレーティングシステム開発シリーズ

- **111**: 未定義、使用しない
- **ビット4-5: (RL0, RL1)** リード/ロードモード。カウンタレジスタにデータを読み込んだり、送信したりします
 - **00**: カウンタ値は、I/O書き込み動作時に内部制御レジスタにラッチされます。
 - **01**: 最下位バイト (LSB) のみ読み出しまたは読み込み
 - **10**: 最上位バイト (MSB) のみ読み出しまたは読み込み
 - **み 11**: LSBの次にMSBを読み出しまたは読み込み
- **ビット6-7: (SC0-SC1)** セレクトカウンタ。それぞれの説明は上記のセクションを参照してください。
 - **00**: カウンタ0
 - **01**: カウンタ1
 - **10**: カウンタ2
 - **11**: 不正な値

PICのインターフェイスと同様に、コマンドのフォーマットを記述するために使用されるいくつかのビットマスクを設定します。それがこちら...

```
#define i86_pit_ocw_mask_bincount 1 //00000001
#define i86_pit_ocw_mask_mode 0xE //00001110
#define i86_pit_ocw_mask_rl 0x30 //00110000
#define i86_pit_ocw_mask_counter 0xC0 //11000000
(i86_pit_ocw_mask_counter)
```

なるほど...PICで設定したICWやOCWに比べると小さいですが、実はこれはもっと複雑です。PICで使われているコマンドは、1ビットの大きさでシンプルです。この操作コマンドワードで使うコマンドはそうではありません。

ここでは、「**コマンドコントロールビット**」が活躍します。これらのビットは、上記の異なるビットマスクのための異なる設定とビットの組み合わせを定義するのに役立ちます。以下にその例を示します。

```
#define i86_pit_ocw_bincount_binary 0 //0 //!!I86_PIT_OCW_MASK_BINCOUNTの設定時に使用します。
#define i86_pit_ocw_bincount_bcd 1 //1
#define i86_pit_ocw_mode_terminalcount 0 //0000 //!!I86_PIT_OCW_MASK_MODEの設定時に使用します。
#define i86_pit_ocw_mode_oneshot 0x2 //0010
#define i86_pit_ocw_mode_ratagen 0x4 //0100
#define i86_pit_ocw_mode_squarewavegen 0x6 //0110
#define i86_pit_ocw_mode_softwaretrig 0x8 //1000
#define i86_pit_ocw_mode_hardwaretrig 0xA //1010
#define i86_pit_ocw_rl_latch 0 //000000 //!!I86_PIT_OCW_MASK_RLの設定時に使用します。
#define i86_pit_ocw_rl_lsbonly 0x10 //010000
#define i86_pit_ocw_rl_msbonly 0x20 //100000
#define i86_pit_ocw_rl_data 0x30 //110000
#define i86_pit_ocw_counter_0 0 //00000000 //!!I86_PIT_OCW_MASK_COUNTERの設定時に使用します。
#define i86_pit_ocw_counter_1 0x40 //01000000
#define i86_pit_ocw_counter_2 0x80 //10000000
```

例を挙げてみましょう。例えば、カウンタ0をバイナリカウントモードの矩形波ジェネレータとして初期化したいとします。これには次のような方法があります。

```
uint8_t ocw=0;
ocw = (ocw & ~I86_PIT_OCW_MASK_MODE) | I86_PIT_OCW_MODE_SQUAREWAVEGEN;
ocw = (ocw & ~I86_PIT_OCW_MASK_BINCOUNT) | I86_PIT_OCW_BINCOUNT_BINARY; ocw = (ocw & ~I86_PIT_OCW_MASK_COUNTER) | I86_PIT_OCW_COUNTER_0;
```

私はこれを簡単にしすぎていると思いますが、あなたはどう思いますか？:p これだけで、**ocw**にはPICに送ることのできる操作コマンドワードが入ります。これらの定数を使用することで、読みやすさを向上させるだけでなく、エラーの可能性を減らすことができることに注目してください。

これがpit.hの全てだと思います。次は、pit.cppに飛び込んでみましょうか。うわああああああああああああああああああああ

pit.cpp:インプリメンテーション

これは、PITミニドライバの大部分を含んでいます。インターフェイスとインプリメンテーションの両方で使用される各ルーチンのインプリメンテーションが含まれています。

pit.cpp:レジスター

ここでは、PITのポート位置を抽象化するための定数を定義しています。

```
#define i86_pit_reg_counter0 0x40
#define i86_pit_reg_counter1 0x41
#define i86_pit_reg_counter2 0x42
#define i86_pit_reg_command 0x43

//!グローバルTickカウント
uint32_t_pit_ticks=0;
```

悪くはありません。**I86_PIT_REG_COUNTER0**、**I86_PIT_REG_COUNTER1**、**I86_PIT_REG_COUNTER2**は、各カウンタのデータレジスタです。PITには3つの内部カウンタがあることを覚えていますか？**I86_PIT_REG_COMMAND** は、コマンドレジスタです。PITを制御・動作させるためには、コマンドレジスタにコマンドを書き込む必要があります。

また、**_pit_ticks**にも注目してください。これはとても特別で重要なグローバルです。

PITのカウンタ0がPICのIRQ0ラインに接続されていることを覚えていますか？つまり、カウンタ0が発火すると、**割り込み要求 (IRQ) 0**が発生します。この要求を処理するために、割り込みハンドラを作成してインストールする必要があります。

割り込みハンドラが行う必要があるのは、システムの**グローバルティックカウント**を更新することです。これが**_pit_ticks**の役目です。

i86_pit_irq()です。PITカウンタ0の割り込みハンドラ

これは、IRQ 0の要求を処理する割り込みハンドラです。カウンタ0が発火するたびに、この割り込みハンドラが呼び出されます。こ

の割り込みハンドラが行うことは、発火するたびにグローバルティックカウントをインクリメントすることだけです。割り込みハン

ドラの一般的なフォーマットに注意してください。

ズ

intstart()は、ハードウェア割り込みを無効にしてスタックフレームを保存し、スタックを欠落させずにタスクに戻るためのマクロです。 **intret()**は、ハードウェア割り込みを無効にしてスタックフレームを復元し、**IRETD**命令を使用してハンドラから戻るためのマクロです。この目的は単純に、現在のスタックが変更されないように保護し、そのスタックをそのままにしてタスクに復帰するためです。これらのマクロは**asm/system.h**で定義されており、カーネルやデバイスドライバの割り込みハンドラで使うことができます。

interruptは、特定のコンパイラでのみ使用される特別な定数です。MSVC++では、**declspec (naked)**と定義されています。これは、コンパイラーが追加したコードを気にしなくてもいいようにするためです。いくつかのコンパイラーはこのキーワードを直接サポートしています (特に 16 ビットコンパイラー)。MSVC++ のように) そうでないものもあるので、私たちが定義しなければなりません。

interruptdone()は、**Hardware Abstraction Layer**で定義された特別なルーチンです。 **割り込み終了** コマンドをPICに送信する役割を担っています。

これは、すべての割り込みハンドラが使用する一般的なフォーマットです。

i86_pit_send_command ():PITへのコマンド送信

これは、PITにコマンドを送信するための重要なルーチンです。これにより、送信先のコマンドポートを隠すことができるので、ポート名を変更する必要がある場合に便利です。 **instcmd**は**OCW(Operation Command Word)**という形式で送られます。

例えば、上記のビットマスクをコマンドコントロールビットを使って、OCWを構築することができます。そして、**i86_pit_send_command()**を使ってOCWをPITに送ることができます。

```
void i86_pit_send_command (uint8_t cmd) {...}
```

// ! ハルに終了したことを伝え

i86_pit_send_data ()および**i86_pit_read_data ()**:カウンタからのデータの送信および読み出し

これらのルーチンは、カウンタの読み書きの際に使用されるポート名を抽象化するために役立ちます。これらのルーチンは、現在のカウンタ値を設定または取得するために使用されます。これらのルーチンが行うことは、正しいポートを得るために、**カウンタ**に渡されたカウンタをテストすることだけです。そして、そのポートを使った単純な読み書き操作を行います。

```
// ! カウンターにデータを送る
void i86_pit_send_data (uint16_t data, uint8_t counter) {...

    uint8_t port= (counter==I86_PIT_OCW_COUNTER_0) ?I86_PIT_REG_COUNTER0 :
        ((counter==I86_PIT_OCW_COUNTER_1) ?I86_PIT_REG_COUNTER1 : I86_PIT_REG_COUNTER2)
        を参照してください。)

    outportb (ポート、データ) です。
}

// ! カウンターからデータを読み込む
uint8_t i86_pit_read_data (uint16_t counter) {...

    uint8_t port= (counter==I86_PIT_OCW_COUNTER_0) ?I86_PIT_REG_COUNTER0 :
        ((counter==I86_PIT_OCW_COUNTER_1) ?I86_PIT_REG_COUNTER1 : I86_PIT_REG_COUNTER2)
        を参照してください。)

    return inportb (ポート) 。
}
```

i86_pit_initialize ():PITを初期化する

では、PITの初期化について説明しましょう。そうですね。初期化の必要がないので、特に話すことはありません。 **irq**は使用する割り込み番号、 **irCodeSeg**は**グローバルディスクリプタテーブル (GDT) のコードセクタ**のオフセットです。

i86_install_irq()ルーチンを使用して、割り込みハンドラ (**i86_pit_irq**) を**割り込み記述子テーブル**にインストールします。これ以降、IRQ 0は**irq**の割り込みハンドラにマッピングされます。IRQ 0にマッピングされていることを確認するために、**irq**はプライマリPICが使用するようにマッピングされたのと同じベースIRQ番号でなければなりません。

i86_pit_start_counter ():内部カウンタを開始する

これは、PITインターフェースの最後のルーチンです。これはカウンターを開始するものです。カウンタを起動したい**カウンタ**に渡します (**I86_PIT_REG_COUNTER0**など)。 **mode**には、カウンタに使用させたい動作モードを指定します (**i86_install_irq**など)。 **freq**は、カウンタに動作させたい周波数レートを指定します。 (**I86_PIT_OCW_MODE_SQUAREWAVEGEN**など)。 **freq**には、 **i86_IDT_DESC_B1F32**, **irCodeSeg**, **i86_pit_irq**;

このルーチンでは、ルーチンに渡されたパラメータに基づいて、**操作コマンド・ワード**を構築します。

```
void i86_pit_start_counter (uint32_t freq, uint8_t counter, uint8_t mode) {...
```



```
if (freq==0)
    を返すことができます。

uint16_t divisor = 1193180 / freq;

// ! 操作コマンドの送信 uint8_t
ocw=0;
ocw = (ocw & ~I86_PIT_OCW_MASK_MODE) | mode;
ocw = (ocw & ~I86_PIT_OCW_MASK_RL) |
I86_PIT_OCW_RL_DATA; ocw = (ocw &
~I86_PIT_OCW_MASK_COUNTER) | counter;
i86_PIT_send_command (ocw) です。

// ! 周波数レートの設定 i86_pit_send_data
(divisor & 0xff, 0);
i86_pit_send_data ((divisor >> 8) & 0xff, 0)。

ティックカウントのリセット
_pit_ticks=0です。
}
```

結論

ここから先は、基本中の基本です。このシリーズでは、プロセッサのモードやアーキテクチャ、プロセッサテーブル、割り込み、割り込み管理など、すべてを網羅しています。これがカーネルの始まりであり、カーネルはここから構築されます。

このチュートリアルでは、PIC、PIT、例外、およびハードウェア割り込み管理のサポートを追加しました。多くの重要なデバイスがハードウェア割り込みを使用しているので、これは重要なステップです。また、ハードウェア割り込みを再び有効にすることができます（プロテクトモードに切り替える前に、ハードウェア割り込みを無効にする必要があったことを覚えていますか）。

次のチュートリアルでは、カーネルそのものに戻ってみましょう。今回は、コンピュータシステムの最も基本的な側面の1つについて説明します。**ページング**と**低レベルのメモリ管理**を行います。これは、私たち自身の**システムAPI**の基礎にもなります。それでは、

またお会いしましょう。）

次の機会まで。

マイク

BrokenThorn Entertainment 社。現在、DoE と [Neptune Operating System](#) を開発中です。質問

やコメントはありますか？お気軽に[お問い合わせください](#)。

あなたも記事の改善に貢献したいと思いませんか？もしそうなら、ぜひ[私に教えてください](#)。



第15章

ホーム



第17章



オペレーティングシステム開発シリーズ

オペレーティングシステム開発 - 物理メモリ

by Mike, 2008

このシリーズは、オペレーティングシステムの開発を一から実演し、教えることを目的としています。

はじめに

Welcome!

このチュートリアルでは、コンピュータシステムの中で最も重要なリソースの1つである「物理メモリ」の管理について説明します。物理メモリです。メモリ情報の取得方法、BIOSの割り込み、メモリマネージャの概念、そして完全な物理メモリマネージャの設計と実装について説明します。

これは誰もがやりたがらないことの一つですが、結果的には作業が非常にやりやすくなります。それでは、このチュートリアルのためのリストを見てみましょう。

- 物理的メモリ
- TLB (Translation Lookaside Buffer)
-) メモリマネージメントユニット (
- MMU) メモリマネージャ
- メモリ情報の保持
- ブートローダからカーネルへの情報の受け渡し 物理的メモ
- リマネージャの設計・開発

それでは、アレイティここでは、ページングや仮想メモリについては説明しません。その代わりに、物理メモリ管理と仮想メモリ管理の概念を完全に分けて考えたいと思います。その理由は、簡単だからです。つまり、どちらか一方に集中すれば、もう一方は不要だからです。ご心配なく。ページングと仮想メモリについては、次のチュートリアルで仮想メモリマネージャの開発を取り上げる予定です。

記憶。より深く見るために

ここでは、いきなりメモリ管理の話に入るのではなく、別のアプローチをとりたいと思います。つまり、メモリ自体が何であるかを理解せずに、メモリ管理が何であるかを理解することはできないということです。つまり、私たちが管理しようとしているのは何なのかを知る必要があるのですね。

このため、まず最初に物理的なメモリとは何かを見ていきます。ほら…。コンピュータの中にある小さなRAMチップのことです。)

いくぞ…!

物理的メモリ

物理的な記憶。概要

物理メモリとは、コンピュータの**ランダムアクセスメモリ (RAM)**内に格納された抽象的なメモリブロックのこと。物理メモリがRAM内にどのように「格納」されるかは、システムが使用するRAMの種類によって異なる。例えば、**DRAM (ダイナミック・ランダム・アクセス・メモリ)**は、各ビットのデータを独自の**コンデンサ**に格納し、定期的にリフレッシュする必要がある。**コンデンサ**とは、限られた時間内に電流を蓄える電子機器のことである。これにより、電流を蓄える(2進1項)ことも、電流を蓄えない(2進0項)こともできる。コンピュータでは、このようにして**DRAM**チップが個々のビットデータを記憶する。

ほとんどの場合、メモリの種類(RAM、SRAM、DRAMなど)によって、プロセッサとのインターフェースに特定のタイプの**メモリコントローラ**が必要となります。**システムバス**。

メモリーコントローラーは、ソフトウェアを使ってメモリーを読み書きする機能を持っています。また、メモリーコントローラーは、**RAM**チップが情報を保持できるように、常にリフレッシュする役割も担っています。

メモリコントローラには、マルチプレクサ回路とデマルチプレクサ回路が搭載されており、正確な**RAM**チップの選択や、アドレスを参照する位置の選択を行います。**アドレスバス**。これにより、アドレスバスを通じてメモリアドレスを送信することで、**プロセッサ**が特定のメモリロケーションを参照することができます。

…ここでソフトウェアの出番です。ソフトウェアは、プロセッサにどのメモリアドレスを読めばいいかを教えてくれます。)

メモリコントローラーは、**RAM**チップ内の場所を順番に選択していきます。つまり、システムの総メモリ量を超える物理メモリの位置にアクセスしても、何も起こらないということです。つまり、そのメモリ位置に値を書き込み、それを読み戻すと、データバスに残ったデータを得ることができるのです。

物理アドレス空間にメモリホールが発生することがあります。これは、例えば、**RAM**チップがスロット1と3にあり、スロット2に**RAM**チップがない場合に起こります。つまり、スロット1の**RAM**に格納されている最後のバイトと、スロット3の最初のバイト-1の間に、存在しないメモリ領域があるということです。これらの場所への読み書きは、メモリを超えて読み書きする場合とほぼ同じ効果があります。この存在しないメモリの場所が、メモリコントローラによってリマップされている場合は、メモリの別の部分に読み書きしている可能性があります。メモリがリマップされていない場合(ほとんどのメモリはリマップされていません)、**存在しないメモリ位置への読み書きは全く何もしません。つまり、存在しないメモリ位置への書き込みは、どこにも何も書き込まれませんが、存在しないメモリ位置からの読み出しは、データバスに残っていたゴミを読み取ることになります。存在しない場所に値を書き込んでも、それを読み返しても同じ値にはならない**ということで、ポインターを使ってメモリを手動で解析し、メモリのどの領域が良いか悪いかを判断する方法が出てきました。しかし、この方法は後述するように危険を伴います。

さて、以上が物理的なメモリの正体です。メモリがどのように各ビットを格納しているかを知れば、バイト、ワード、ワード、クォード、タバイトなどがどこから入ってくるかがわかるでしょう。これらの中で最も重要なのは、プロセッサがアクセスできる最小のデータである「**バイト**」です。しかし、プロセッサはどのようにしてバイトがメモリのどこにあるかを知のでしょうか?そこで登場するのが「**物理アドレス空間**」です。それでは見ていきましょう。)

PAS (Physical Address Space) の略。

物理的なメモリ(RAM)に格納されている8ビットのデータ(つまり1バイト)を参照するために、プロセッサが使用する(メモリコントローラが変換する)アドレス空間です。**メモリーアドレス**とは、**メモリーコントローラー**が1バイトのデータに対して選択した番号のことです。例えば、メモリ

メモリアドレス0は最初の8ビット、メモリアドレス1は次の8ビットというように、物理的なメモリを参照することができます。これらの**メモリアドレス**と、そのメモリアドレスが参照する実際のメモリを配列したものが「**物理アドレス空間**」である。

物理アドレス空間は、システム・アドレス・バスを介して**プロセッサ**からアクセスされます（[第7](#)で説明しましたね）。

さて、プロセッサはメモリのバイトを参照するためにアドレスを使うことができます。通常、アドレスは0から始まり、メモリの各バイトごとに増加していきます。このように簡単ですね。しかし、これだけでは、ソフトウェアがどのようにメモリにアクセスするかを説明できません。確かに、**プロセッサ自体はメモリを参照する方法を持っていますが、ソフトウェアはそうではありません**。プロセッサは、その必要性に応じて、ソフトウェアがメモリを参照する方法を提供するための特定の方法を提供する必要があります。え、なに？メモリのアドレスやアクセス方法の違いですね。

アドレッシングモード

アドレス指定モードとは、ソフトウェアが**物理アドレス空間**にアクセスする方法を管理するために、プロセッサが作成する抽象化された機能です。通常は、ソフトウェアがプロセッサのレジスタを設定して、プロセッサがメモリを参照する方法を知るようにします。ここでは、「**セグメント：オフセット**」と「**ディスクリプター：オフセット**」の2つを紹介します。

メモリへのアクセス方法を許可するために、プロセッサがソフトウェアに与えるインターフェースです。

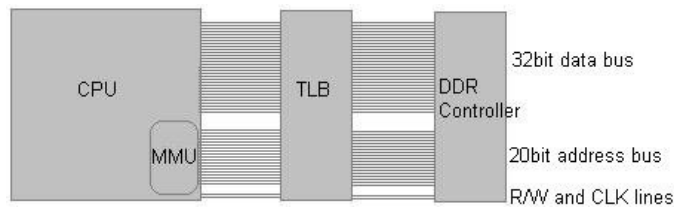
[第4章](#)ではセグメント：オフセットのアドレッシングモードを、[第8章](#)ではディスクリプター：オフセットのメモリアドレスアドレッシングモードを取り上げました。

記憶のしくみ。詳細

さて、ここからは新しい視点でメモリを見ていきましょう。これまでに、メモリとは何か、アドレス空間、アドレッシングモードなどの詳細を説明してきました。さて、これらをまとめてみましょうか。

ここに書かれている内容は必要のないものも多いのですが、念のために記載しておきます。ここに書かれていることが理解できなくても、あまり気にしないでください。

[第7章](#)では、コンピュータ・システムの基本的な概要とシステム・アーキテクチャについて説明しました。これまで、プロセッサの**システムバス**は、システムが物理的なRAMを制御する方法を提供するために使用される**メモリコントローラ**に接続されています。このような感じです。



その通りです。これが、物理的なRAMがシステムの他の部分と接続し、通信する方法です。上の図では、「**DDR Controller**」がメモリコントローラです。

TLB (Translation Lookaside Buffer) は、メモリコントローラとプロセッサの間にあります。これにより、**システムバスは、アドレスバス、データバス、コントロールバス**を介して、これら3つのバスを接続する。制御バスの中で今重要なのは、**RWライン**と**CLKライン**の2本だけだ。

TLBは、ページングが有効な場合にのみ使用されます。このため、後でもう少し詳しく見てみましょう。

では、物理的なメモリロケーションにデータを書き込むときには、実際に何が起ころのでしょうか。**書き込みの際には、プロセッサはRWピンをハイレベル（論理的に1）にします**。これにより、接続しているデバイスに書き込み操作が行われることを伝えます。プロセッサは、**IOコントロールラインをロー（論理的な0）にリセット**します。これにより、IOサブシステムはコマンドを無視し（IN/OUTポートの命令ではないことを意味する）、メモリコントローラのためのコマンドを実行します。プロセッサは、書き込み先のアドレスをアドレスバスに、書き込みデータをデータバスにコピーします。これらのラインはメモリコントローラに間接的に接続されているため、メモリコントローラはそれが書き込み操作であることを認識することができる。メモリコントローラは、アドレスバス上のメモリアドレスをデマルチプレクサ回路で変換し、使用する**RAMチップ**を見つけ、リニアオフセットバイトをRAMチップのメモリ空間に入れるだけでよい。その後、メモリコントローラは、データバスからこの場所にデータをコピーし、次のクロック信号でメモリの状態を更新します。

読み出し時には、書き込み時とほぼ同様の処理が行われます。ただし、**RWラインがLow**に設定され、読み出し動作を示します。また、メモリコントローラは、メモリアドレスをRAMチップ内のオフセットに変換した後、その位置に格納されているデータをコピーして、プロセッサ用のデータバスに配置します。そして、メモリコントローラは、次のクロック信号でメモリの状態をリフレッシュする。

CLK信号は、リードとライトによるアドレスとデータ値の交換を同期させるために使用されます。メモリチップとの通信は、CLKラインが論理1（ハイに設定）のときに開始されます。CLKラインがハイになっている間、アドレスはアドレスラインに置かれ、R/Wラインは書き込みの場合はハイに、読み出しの場合はローになります。

実行中、プロセッサはメモリコントローラとの間で読み書きを行うために、常にクロックラインをハイ/ローに切り替えます。

ページングが禁止されている間、TLB自体は全く何もしません。メモリを読み書きするときには、TLBはまったく使われないことに注意してください。

物理メモリマネージャ

ご存知のように、メモリの管理は非常に重要です。すべてのデータとコードは、同じ物理アドレス空間を共有しています。もし、より多くのデータやプログラムを読み込んで作業しようとする、それを可能にするためのメモリ管理方法を何らかの方法で見つける必要があります。

この段階では、私たちのカーネルは、コンピュータ内のすべてのハードウェアとメモリを完全に制御することができます。これは素晴らしいことですが、同時に悪いことでもあります。メモリのどの領域が現在使われているのか、どの領域が空いているのかを知るすべがありません。プログラムの破損、データの破損、メモリがどのようにマッピングされているか分からない、トリプルフォールトやその他の例外エラーなどです。予測できない結果になるかもしれません。

そのため、物理メモリを効果的に管理することは非常に重要です。もっと詳しく見てみましょう。

メモリの検出

アブストラクト

まず、コンピュータシステムに搭載されているRAMの容量を調べる必要があります。これにはさまざまな方法があります。あるシステムではうまくいく方法もあれば、そうでない方法もあるでしょう。

メモリ量の確保は、システムに大きく依存します。具体的には、マザーボードのチップセットに依存します。初期化の際、BIOSはメモリコントローラからメモリ情報を取得し、検出されたメモリで動作するようにチップセットを構成します。このため、**OSはシステムBIOSを経由してメモリ情報を取得する必要があります。**でも、プロテクトモードではBIOSを使えないじゃなかったっけ？そうなんです。その代わり、他の方法で情報を得る必要があります。ブートローダでしょうか？

システム内のメモリ量を確保するために、他の方法もあることを指摘しておきます。例えば、CMOS、PnP、SMBiosなどです。しかし、正しい量を得ることを保証する唯一の方法は、それを設定するデバイスからです。BIOSです。

最後に、すべてのPCには、メモリマップされたハードウェアやBIOS ROMなどの追加デバイスのために、4GB以下のメモリ領域が必要となります。これを回避する方法は後ほど紹介しますが、ご安心ください。)

1MB以下のメモリを **"Low Memory"**、別名 **"Conventional Memory "**と呼びます。1MB以上のメモリは**Extended Memory**と呼ばれます。

この点を考慮して、いくつかの素晴らしいBiosの割り込みを見てみましょう。

Bios:メモリサイズの取得

以下のルーチンはすべて、このチュートリアルのある2ndステージのブートローダ内のデモの**memory.inc**にあります。

BIOS INT 0x12 - Get Memory Size (Conventional Memory)

戻る

CF = 成功したらクリア
AX = コンベンショナル・メモリのKB数
AH = エラー時のステータス (0x80: 無効なコマンド; 0x86) サポートされていない機能

これが一番簡単な方法だと思います。この割り込みは、BIOSデータエリア（物理アドレス0x413のワード）にある値を返します。WORDサイズの値を返すため、0xFFFF（10進数で65535）までの制限があります。つまり、64KB以下のメモリしか検出できません。このため、64KB以上のメモリを持つシステムでは、正しいサイズを返すことができません。そのため、この方法は使用しません。

この方法は、完全なメモリサイズを返さないかもしれませんが、すべてのPCとは言わないまでも、ほとんどすべてのPCで動作することが保証されている唯一の方法です。

```
BiosGetMemorySizeで
す。    int      0x12
        jc      .error
        test    ax,
        je      ax                サイズが0の場合
        cmp     .error
        je      ah, 0x86          サポートされていない機能
        cmp     .error
        je      ah, 0x80          無効なコマンド
        ret     .エラー
.エラー
になり  mov     軸, -1
ます。   ret
す。     rea
        t
```

BIOS INT 0x15 Function 0x88 - Get Extended Memory Size

戻る

CF = 成功したらクリア
AX = 1MBの物理アドレスから始まる連続したKBの数
AH = エラー時のステータス (0x80: 無効なコマンド; 0x86) サポートされていない機能

この割り込みは、AXのKB拡張メモリの量を返します。16ビットのレジスタを使用しているため、64MBまたは0xFFFFF（65535）を返すことに制限されています。Windowsの一部のバージョンでは、この関数は代わりに15MBを返す場合があります。

```
BiosGetExtendedMemorySizeです。
movab   ax, 0x88
le      int      0x15
        jc      .エラー
        test    ax, ax            サイズが0の場合
        jz      .エラー
        cmp     ああ, 0x86        サポートされていない機能
        jz      .エラー
        cmp     ああ, 0x80        無効なコマンド
        jz      .エラー
        ret
.エラー
になり  movab   軸, -1
ます。   le
す。     レット
```

BIOS INT 0x15 Function 0xE881 - Get Memory Size For > 64 MB Configurations (32 Bit)

戻る

CF = 成功したらクリア
EAX = 1MBから16MBまでの拡張メモリ（単位：KB） EBX
= 16MB以上の拡張メモリ（単位：64KB） ECX = 1MBから
16MBまでのコンフィグレーションメモリ（単位：KB）

この割り込みは、拡張レジスタ（EAX/EBX/ECX/EDX）を使用することを除けば、INT 0x15 Function 0xE801と全く同じです。

BIOS INT 0x15 Function 0xE801 - Get Memory Size For > 64 MB Configurations

戻る

CF = 成功したらクリア
EAX = 1MBから16MBまでの拡張メモリ（単位：KB） EBX
= 16MB以上の拡張メモリ（単位：64KB） ECX = 1MBから
16MBまでのコンフィグレーションメモリ（単位：KB）
EDX = 16MB以上のメモリを64KBのブロックで構成

これは、私がよく使う方法です。この割り込みは、Windows NTとLinuxの両方で起動時に使用され、INT 0x15関数0xe820がサポートされていない場合、メモリサイズを検出します（Get System Memory Map）。これについては後で見ます。この方法は1994年頃から使われているので、古いシステムではこの方法をサポートしていないかもしれません。

Extended Memory」と「Configured Memory」の値は、ほとんど同じです。BIOSによっては、EAXとEBX、ECXとEDXのいずれかに格納する場合があります。つまり、BIOSによっては、EAXとEBXは使うが、ECXとEDXはそのままにしておく場合があります。他のBIOSは全く逆のことをするかもしれません。標準規格に感謝します。:)あ、そうですか...すみません ;)

この方法の典型的な使い方は、BIOSを呼び出す前に、まずすべての汎用レジスタをヌルにすることです。こうすることで、BIOSを呼び出した後に、レジスタがヌルかどうかをテストすることができ、使用するレジスタのペアを知ることができるのです。EAX/EBXとECX/EDXのどちらを使うべきかがわかります。

```
;-----  
; 64Mを超える設定の場合のメモリサイズ の取得  
1MBから16MBまでの間にあること  
bx=16MB以上の64Kブロックの数  
エラー時には bx=0, ax= -1 と になります。  
;-----  
  
BiosGetMemorySize64MBです。  
    ユ  
    プッシュ    edx  
    ユ  
    xor        ecx, ecx           ;すべてのレジスターをクリアします。これは後のテストに必要です  
    xor        edx, edx  
    movab     ax, 0xe801  
    le  
    int       0x15  
    jc        .エラー  
    cmp       ああ、0x86         サポートされていない機能  
    jz        .エラー  
    cmp       ああ、0x80         無効なコマンド  
    jz        .エラー  
    jcxz      .use_ax             バイオスはax,bxまたはcx,dxに格納している可能性があります,cxが0であ  
                                るかどうかをテストします。  
                                そうではないので、メモリサイズを含んでいるはずです。  
.use_ax:movab     ax, cx  
        le  
        movab   bx, dx  
        le  
  
.エラー   ポップ    edx           メモリサイズはすでにaxとbxに入っているのです、それを返  
になりま  mov     軸、-1  
す。      mov     ebx, 0  
        呼び出  edx  
        呼び出  ecx  
        pop     ecx  
        ret
```

このルーチンが返す結果に注目してください。システム内のKBの量を得るためには、いくつかの計算をする必要があります。EBXには、64KBのメモリブロックの数が含まれています。これを64倍すると、実質的にEBXの値が16MB以上のKB量に変換されます。その後、EAXで返された数字にこの数字を足すだけで、1MB以上のKB数がわかります。1メガバイトの中には1024KBあるので、この数字に1024を加えれば、システム内のKBの総量になります。

メモリを手動で調べる

メモリを手動で探査するとは、メモリに直接アクセスしてポインタからメモリを手動で検出することです。この方法では、すべてのメモリを検出できる可能性があります。最も危険な方法でもあります。私たちの知らないところで、メモリの領域を別の用途に使っているデバイスがあるかもしれないことを忘れないでください。また、メモリマップドデバイスやROM BIOSなど、メモリを使用するデバイスがあるかもしれません。また、物理的なアドレス空間内のメモリホールも考慮に入れていません。

メモリを直接調べるということは、存在しないメモリに対して読み書きしても何も起こらないことに由来する。つまり、存在しない物理メモリのアドレスに書き込んでも、エラーにはなりません。しかし、同じ場所から再び読み出そうとすると、データベースに残っていた値が全くのランダムなゴミとなることがあります。

このように、メモリを調べるには、1k（くらい）のメモリごとにループに入ればいいのです。ポインタを使ってメモリの位置を読み書きします。ポインタから読み取った値が無効な値になるまで、ポインタをインクリメントし続ける（つまり、メモリの別の場所から読み取る）。

このメソッドのためにちょっとしたデモコードを作成するかもしれませんが、多くの問題があるため、おそらく使用することはないでしょう。しかし、メモリを検出する方法としては、最も安全ではない方法であり、予想外の結果を引き起こす可能性があるため、この方法を含めることにしました。ご自身の責任でお使いください。

メモリマップの取得

イッパイアッテナこれで、システム内のメモリ量がわかりました。でも、ちょっと待って！このメモリのすべてが使えるわけではないんですね。

そこで登場するのが「メモリマップ」です。メモリマップは、メモリのどの領域が何に使われているかを定義します。これを使えば、どの領域を使えば安全なのかを把握することもできます。

BIOS INT 0x15 Function 0xE820 - Get Memory Map

2021/11/15 13:09
入力

オペレーティングシステム開発シリーズ

EAX = 0x0000E820

EBX = 継続値、またはマップの先頭から開始する場合は0 ECX
= 結果を格納するバッファのサイズ (必ず ≥ 20 バイト)

EDX = 0x534D4150h ('SMAP')
ES:DI = 結果のバッファ

戻る

CF = 成功したらクリア
EAX = 0x534D4150h ('SMAP')
EBX = コピーする次のエントリのオフセット、または完了した場合は0 ECX = 返される実際の長さ (バイト)
ES:DI = バッファフル
エラーの場合、AHはエラーコードを含む

アドレスレンジ記述子

この割り込みで使用するバッファは、以下のフォーマットに従ったディスクリプターの配列です。

フィールド名	型	サイズ	説明
<code>.baseAddress</code>	レスキュー	1	アドレスレンジのベースアドレス
<code>.長さ</code>	レスキュー	1	アドレス範囲の長さ（バイト）
<code>.タイプ</code>	resd	1	アドレスレンジの種類
<code>.acpi_null</code>	resd	1	予約

アドレスレンジの種類

この機能で定義されているアドレスレンジの種類を以下に示します。1：使用

- 可能なメモリー
- 2: 予約済み、使用しないでください。(例：システムROM、メモリマップド
- デバイス) 3: ACPI Reclaim Memory (ACPIテーブルを読み込んでOSが使用可能)
- 4: ACPI NVS Memory (OSはNVSセッション間でこのメモリーを保存する必要がある) その
- 他の値は未定義として扱うべきである。

メモリマップの取得

このインタラプトは少し複雑に見えるかもしれませんが、悪くはありません。

まず、この割り込みが必要とする**入力**を見てみましょう。もちろん、**AX**にはファンクション番号 (**0xe820**) を入れています。しかし、一部のBIOSでは、**EAX**の上半分がゼロであることが**必要**です。このため、ここでは**AX**ではなく**EAX**を使用する必要があります。

また、EDXには「SMAP」の値が含まれていなければならないことにも注意してください。これは別の要件です。一部のBIOSでは、割り込みを呼び出した後にこのレジスタをゴミ箱に入れてしまうことがあります。

なるほど…。この割り込みを実行すると、BIOSはメモリマップの1つのエントリを返します（このエントリにはフォーマットがあります。上記の「アドレスレンジ記述子」を参照してください）。割り込みを実行した後、EBXが0でない場合、メモリマップにはさらに多くのエントリがあることになります。マップ内の各エントリをループする必要があります。エントリの長さが0の場合、そのエントリには何もないのでスキップして、最後まで次のエントリに進みます。

このルーチンは、上記で定義したMemoryMapEntry構造体を使用して、バイオから取得したエントリから情報を取得します。

```

; バイオスからメモリマップを取得
; /in es:di->エントリーのためのデスティネーション・バッファ
; /ret bp=エントリー数
;-----
BiosGetMemoryMap:
    mov     ebx, 0
    mov     bp, bp
    mov     edx, 'PAMS'
    le     eax, 0xe820
    le     ecx, 24
    int     0x15
    jc     .エラー
    cmp     eax, 'PAMS'
    jne     .エラー
    test    ebx, ebx
    jnz     .エラー
    jmp     .スタート

    mov     edx, 'PAMS'
    le     ecx, 24
    mov     eax, 0xe820
    mov     ecx, [es:di + 0x15]
    test    ecx, ecx
    jnz     .スタート
    jcxz    .skip_entry
    mov     ecx, [es:di + 0x15 + MemoryMapEntry.length + 4]
    jecxz   .skip_entry

    good_entry:
    .skip_entry:
    .エラー

```

inc	bp	エントリカウンタの増加
追加	ディ、24	バッファ内の次のエントリへのポイントdi
cmp	ebx, 0	ebxの戻り値が0であれば、リストは終了です
jne	.next_entry	次のエントリへ
jmp	.done	

マルチブート仕様

私は、マルチブートの仕様をすぐに取り上げるつもりはありません。将来的には可能性があります、今ではありません。しかし、ブートローダの内部でBIOSから得た情報をカーネルに渡す方法が必要です。これはどのような方法でも可能です。マルチブート仕様では標準的なブートタイム情報構造が定義されていますし、私たちがマルチブート仕様を完全にサポートするかどうかともわかりませんので、同じ構造を使ってはどうかと考えました。

また、他のブートローダ（GRUBなど）を使用する場合は、そのブートローダでカーネルを起動することも可能です。

とにかく、この仕様自体がかなり大きいので、メモリ管理のチュートリアルでカバーするのは良いアイデアではありません ;)そこで、必要な情報の受け渡しに利用できるように、十分な内容を説明します。

アブストラクト

Multiboot仕様は、オペレーティングシステムのカーネルをロードして実行するためのブートローダーの規格を記述するための規格リストです。この仕様では、オペレーティングシステムが制御を開始する前にマシンが置かなければならない標準的な状態が記述されているため、複数のオペレーティングシステムを簡単に起動することができます。また、ブートローダからカーネルへの情報の受け渡し方法や内容も含まれています。

今回は、マルチブートの完全な仕様については取り上げません。しかし、カーネルが実行されたときにマシンの状態がどうなっていなければならないかを見てください。また、ブートローダからカーネルに渡される情報を含む **Multiboot 情報構造**についても少し見てみましょう。また、この構造体を使ってブートローダのメモリ情報を渡すことも考えています。

マシンの状態

マルチブート仕様では、32ビットのOSを起動する（つまりカーネルを実行する）際に、マシンのレジスタを特定の状態に設定する必要があります。具体的には**カーネルを実行する際には、レジスタを以下の値に設定してください。**

- EAX - マジックナンバー。 **0x2BADB002**でなければなりません。EBX - **マルチブート情報構造体**の物理アドレスが含まれてい
- す。
- CS - オフセットが`0`、リミットが`0xFFFFFFFF`の32ビットの読み取り/実行コード・セグメントでなければなりません。正確な値は未定義です。
- DS,ES,FS,GS,SS - オフセットが`0`でリミットが`0xFFFFFFFF`の32ビットのリード/ライト・データ・セグメントでなければなりません。正確な値はすべて未定義です。
- A20ゲートが有効であること
- CR0 - ビット31(PG)はクリア(ページング無効)、ビット0(PE)はセット(プロテクトモード有効)する必要があります。その他のビットは未定義

その他のレジスターはすべて未定義です。これらのほとんどは、既存のブートローダですで行われています。唯一追加しなければならないのは、EAXレジスタとEBXの2つだけです。

私たちにとって最も重要なものはEBXに格納されています。ここには、マルチブート情報構造体の物理アドレスが格納されています。早速見てみましょう。

マルチブート情報構造

この構造体は、マルチブート仕様の中でも最も重要な構造体の一つです。この構造体の情報は、EBXレジスタからカーネルに渡されます。これにより、ブートローダがカーネルに情報を渡すための標準的な方法となります。

これはかなり大きな構造ですが、悪くはありません。これらのメンバーのすべてが必要なわけではありません。仕様では、オペレーティングシステムは、構造体のどのメンバーが存在し、どのメンバーが存在しないかを決定するために、**flags**メンバーを使用しなければならないとされています。

```
struct multiboot_info
{
    .flag      1      必要
    .grub     1      のメモリサイズ。 flags[0]が設定されている場合に表示されます。
    .memo     1
    ryLoresd  1      ブートデバイス。 flags[1]が設定されていれば存在する
    .memo     1      カーネルのコマンドライン。 flags[2]が設定されていれば存在する
    ryHiresd  1      カーネルと一緒にロードされたモジュールの数 flags[3]が設定されていれば存在する
    .boot     1
    Deviceresd 1
    .cmdL     1      flags[4]またはflags[5]が設定されている場合、シンボルテーブル情報が表示されます。
    ineresd   1
    .mods     1
    _countresd 1      メモリマップ。 flags[6]が設定されている場合に存在する
    .mods     1
    _addrresd 1      最初のドライブ構造体の物理的アドレス flags[7]が設定されていれば存在する
    .syms     1
    0resd     1      フラグ[8]が設定されている場合に表示されます。
    .syms     1      ; ブートローダ名。 flags[9]が設定されている場合に存在する。
    .vbe_control_info resd 1      アドバンストパワーマネージメント (apm) テーブル flags[10]が設定されている場合に存在
    .vbe_mode_info resw 1      する
    .vbe_mode     resw 1      flags[11]が設定されている場合は、ビデオ・パイオ・エクステンション (vbe) 。
    .vbe_interface_seg resw
    .lengthresd 1
    .vbe_interface_offset resw
    .addrresd 1
    .drive_list_ptr resd resw
    .drive_list resw
1 endstruct
    .driv
    es_addrresd
    .conf
    ig_tableresd
    .ブートローダ名 resd
    .apm_
    tableresd
```

この構造には多くの情報が含まれています。**memLo**と**memHi**には、BIOSから検出したメモリ量が入ります。 **mmap_length**と**mmap_addr**は、BIOSから取得したメモリマップを指します。

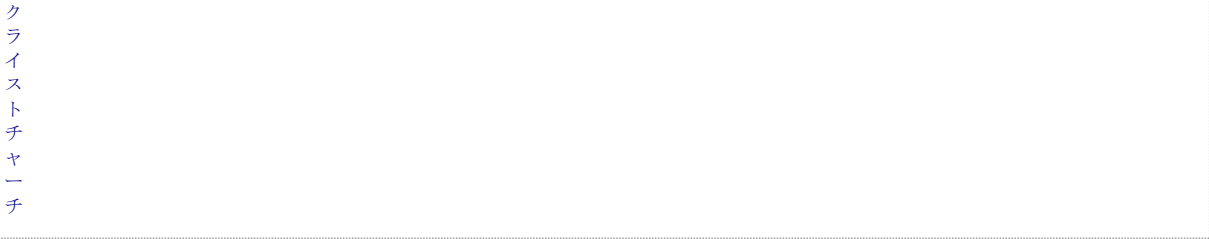
2021/11/15 13:09

オペレーティングシステム開発シリー

これで完成です。これで、カーネルにメモリ情報（その他の情報も含む）を渡す良い方法ができました。

```
mov    eax, 0x2BADB002      マルチブートの仕様では、eaxは次のようになります
mov    ebx, 0
      .
      edx, [ImageSize] (イ
      メージサイズ)

ブッ   ドワード
シュ   boot_info ebp      カーネルの実行
・コ   esp, 4
ール
・ア
ド
```



...そして、我々のカーネルの中にも。

カーネルの `multiboot_info` 構造体は `boot_info` の構造体と同じです。この設定のおかげで、カーネルが行うべきことは、`boot_info` を通じてメモリ情報 (および `multiboot_info` から渡される情報) にアクセスすることだけです。いいでしょう？

さて、`Bios` からメモリ情報を得て、それをカーネルに渡したら、カーネルはそれを物理メモリマネージャに使えるようになります。そう、いよいよ物理メモリマネージャを開発する時が来たのです。

物理的なメモリ管理

すでに多くのことをカバーしてきたと思いませんか？ここでは、`BIOS` からメモリ情報を取得する方法と、マルチブート情報構造を使用してこの情報をカーネルに渡す方法を見てきました。これにより、カーネルはいつでもこのメモリ情報を取得できるようになります。しかし、最も重要なトピックをまだ取り上げていません。このメモリの**管理**です。もう少し詳しく見てみましょう...

メモリ管理。概要

さて、メモリを管理する方法が必要であることはわかっています。そのためには、当然ながら、メモリがどのように使われているかを追跡する方法が必要です。しかし、メモリ上のすべてのバイトについて記録することは不可能です。どうすればメモリ不足にならずに、メモリ内のすべてのバイトの情報を保存できるでしょうか？そのため、別の方法を考えなければなりません。

ここで気をつけなければならないのは、残りのメモリを管理するために使用するデータ構造が、メモリの合計サイズよりも小さいということです。例えば、バイトの配列を利用することができます。各バイトには、より大きなメモリブロックの情報を格納することができます。これがメモリ不足にならないための唯一の方法なのです。

メモリの「ブロック」のサイズは、実現可能で効率的なサイズでなければなりません。この方法を使えば、物理アドレス空間を「ブロック」サイズのチャンクに分割することができます。メモリを割り当てる際には、バイトを割り当てるのではなく、メモリブロックを割り当てます。むしろ、メモリのブロックを割り当てるのです。これが、**物理メモリマネージャの役割**です。

物理メモリマネージャの目的は、コンピュータの物理アドレス空間をブロックサイズのメモリチャンクに分割し、それらを割り当てたり解放したりする方法を提供することです。

`x86`アーキテクチャでは、ページングが有効な場合、各ページは**4KB**のメモリブロックを表します。このため、物理メモリマネージャの各メモリブロックのサイズも**4KB**とし、シンプルな構成としました。

設定方法

さて...物理メモリマネージャが重要であることはわかっています。また、物理メモリマネージャは、物理アドレス空間を分割し、どのメモリブロックが使用されているか、あるいは利用可能かを追跡する必要があることもわかりました。しかし、ちょっと待ってください。カーネルには、メモリを管理するためのメモリ領域が必要なのです。メモリを確保する前に、どうやってメモリ領域を確保するのでしょうか？

できません。このため、唯一の方法は、メモリ内の場所へのポインタを使用することです。この場所は、`BIOS`や**BDA (Bios Data Area)**、カーネル自体と同じように、単に予約されたメモリと考えてください。これを予約されたメモリのどこかに貼り付けたいのですが、カーネルの最後ではどうでしょうか？その後、データ構造の中で、この領域 (カーネル自体と一緒に) を予約済みとマークして、何も触れないようにします。

すばらしい!これで、メモリ内のある場所へのポインタができたので、各ブロックを追跡するために必要な情報をメモリ内に格納することができます。でも...どうやって?つまり、私たちが持っているのはポインタだけなのです。このポインタが指すデータは、メモリの領域を有効に使うために、何らかの使用可能な構造になっていなければなりません。どうすれば、物理的なメモリのすべてを管理する構造を作ることができるのでしょうか？

これには**2つ**の一般的なソリューションがあります。スタックまたはビットマップです。

スタックベースの割り当て

ビットマップベースの割り当て

これは最も簡単な実装方法です。物理メモリマネージャが知る必要があるのは、メモリブロックが割り当てられているかどうかだけです。割り当てられていれば、バイナリビット**1**を使うことができます。割り当てられていない場合は、バイナリビット**0**を使用します。つまり、メモリの各ブロックに対して、割り当てられているかどうかを**1つ**のビットで表します。これが、今回使用する方法です。しかし、物理メモリマネージャは、他の方法 (スタックベースのアプローチなど) も可能なように設計されています。

ビットマップアプローチは、サイズが非常に効率的です。各ビットがメモリのブロックを表しているので、このビットマップアプローチを使った**1つ**の**32ビット**が**32ブロック**を表しています。**32ビット**は**4バイト**なので、**4バイト**のメモリで**32ブロック**のメモリを監視できることになります。

この方法は、メモリブロックを割り当てるたびにビットマップを検索して空きブロック（最初のビットが0）を探す必要があるため、少し時間がかかります。

PMM(Physical Memory Manager)の開発

今度のデモコードでは、物理メモリマネージャ全体が **mmngr_phys.h** と **mmngr_phys.cpp** に記述されています。また、更新された第2ステージのブートローダを見て、ブートローダからカーネルにどのようにメモリ情報が渡されるか、カーネルがどのようにPMMを初期化するかを確認するのもよいでしょう。

グローバルとコンスタント

お気づきかもしれませんが、私は「魔法の数字」が好きではありません;)そのため、これらの数値はすべて、より読みやすい定数の後ろに隠すようにしています。

```

//!8ブロック/バイト
#define PMMNGR_BLOCKS_PER_BYTE 8

//!ブロックサイズ (4k
#define PMMNGR_BLOCK_SIZE      4096

//!ブロックアライメント
#define PMMNGR_BLOCK_ALIGN      pmmngr_block_size

```

これらは、コードの読みやすさを向上させるためのものです。PMMでは、**Memory Block**という抽象的な概念を作ります。メモリブロックのサイズは4096バイト (4K) です。これは、ページングを有効にしたときの1ページのサイズでもあるので、重要です。

また、すべてを把握するために、いくつかのグローバルが定義されています。

その中で最も重要なのが **_mmngr_memory_map** です。これは、すべての物理メモリを追跡するために使用するビットマップ構造へのポインタです。 **_mmngr_max_blocks** は、利用可能なメモリブロックの量を表します。これは、(ブートローダーからBIOSから取得した) 物理メモリのサイズを **PMMNGR_BLOCK_SIZE** で割ったものです。 **_mmngr_used_blocks** は現在使用されているブロックの量、 **_mmngr_memory_size** は参考までに物理メモリの量をKBで表す定数です。

```

static uint32_t _mmngr_used_blocks=0;

```

メモリのビットマップ

いいですか!? **_mmngr_memory_map** は、 **uint32_t** のポインタ... ですよね? そうですね、もちろん... そんな感じです。むしろ、「一連のビットへのポインタ」と考えた方がよいでしょう。各ビットは、そのブロックが割り当てられていない (使用可能) 場合は0、予約されている (使用中) 場合は1になります。この配列のビット数は **_mmngr_max_blocks** です。つまり、各ビットは1つのメモリブロックを表しており、これは4KBの物理メモリに相当します。

このことを知っていれば、ビットマップでやるべきことは、ビットをセットしたり、ビットをアンセットしたり、ビットがセットされているかどうかをテストしたりすることです。それでは見てみましょう。

mmap_set () - ビットマップのビットを設定する

私たちがやりたいことは、メモリマップを整数の配列ではなく、ビットの配列として考えられるようにすることです。これはそれほど難しいことはありません。

ビットは0から **max_x** までの値を **set** (1でbit) で設定したいビットです。このビットを32で割ると、次のような整数のインデックスが得られます。ビットが入っている **_mmngr_memory_map**。

```

_mmngr_memory_map[bit / 32] |= (1 << (bit % 32));

```

このルーチンを使用するには、設定したいビットを指定して呼び出すだけです。 **mmap_set(62)** は、メモリマップのビット配列の62番目のビットを設定します。

mmap_unset () - ビットマップのビットをアンセットする

これは上記のルーチンと非常によく似ていますが、代わりにビットをクリアします。

mmap_test () - あるビットが設定されているかどうかをテストする

このルーチンは、ビットが1であれば真を、0であれば偽を返すだけです (512より32より小さく、非常によく似ていますが、ビットを設定する代わりに、マスクとして使用し、その値を返します。

```

inline bool mmap_test (int bit) {
    _mmngr_memory_map[bit / 32] & (1 << (bit % 32)) を返します。
}

```

これですべてです。ビットマップ内のビットをセット、アンセット、テストする方法ができたので、ビットマップ内の空きビットを検索する方法が必要になります。このビットは、使用可能な空きメモリブロックを見つけるために使用されます。

mmap_first_free () - ビットマップ内の最初の空きビットのインデックスを返す

このルーチンは少し複雑です。メモリビットマップのビットをセット、クリア、テストする方法があります。例えば、メモリブロックを割り当てたいとします。空いているメモリブロックはどうやって見つけるのでしょうか？ビットマップのおかげで、必要なのはセットされていないビットを探してビットマップを横断することだけです。これはそれほど複雑ではありません。

pmmngr_get_block_count()は、このシステムのメモリブロックの最大数を返します(これはビット配列のビット数でもあることを覚えていますか?)これを32(32ビット/ワード)で割って、ビットマップ内の整数の量を求めます。言い換えれば最外周のループは、配列内の各整数を単純にループしているだけです。

そして、そのワードが0で設定されているかどうかを判断するループをビットではなくワードで行うのは、その方がはるかに効率的で速いからです。0xffffffffでないことを確認してからテストします。0xffffffffであれば、次のワードに進みます。もしそうでなければ、ビットをクリアしなければなりません。その後、そのワードの各ビットを調べて空きビットを見つけ、その物理フレームアドレスを返します。{//! ドワードの各ビットをテストする

物理メモリマネージャには、このルーチンの別バージョンである **mmap_first_free_s()** があり、特定のサイズのフレームの最初の空きシリーズのインデックスを返します。これにより、単一のブロックではなく、特定の領域のメモリリンクが空いていることを保証することができます。このルーチンは少々トリッキーなので、もし読者の皆様がコードを理解できないのであれば、このチュートリアルで喜んで詳細を説明したいと思います。

フィジカルメモリアロケータ

私たちは今、記憶を管理する方法を持っています。待って、何?そうなんです。この方法では、ビットマップの各ビットが4KBの物理メモリを表していることを覚えておいてください。最初のメモリブロック (最初の4k) を割り当てたい場合は、ビット0を設定します。2つ目の4kを割り当てたい場合は、ビット1を設定するだけです。これをメモリの最後まで続けます。これにより、4kブロックのメモリを扱うだけでなく、どのメモリが現在使われているのか、予約されているのか (ビットが1)、使用可能なのか (ビットが0) を知ることができます。これらはすべて、上記の3つのシンプルなるルーチンと、ビットマップ配列によって提供されます。かっこいいでしょう?

あとは、実際のアロケーションとデアロケーションのルーチンが必要です。しかし、その前に、ビットマップ領域をBIOSメモリマップのものに初期化する必要があります。さらにその前に、カーネルが物理メモリマネージャが使用するための情報を提供する方法を提供する必要があります。では、見てみましょう。

pmmngr_init () - 物理メモリマネージャを初期化する

このルーチンは、物理メモリマネージャ (PMM) を初期化するためにカーネルから呼び出されます。 **memSize**は、PMMがアクセスを許可される最大メモリ量です。 **bitmap**は、PMMがメモリのビットマップ構造に使用する場所です。もうひとつの重要な点は、 **memset()**コールを使用してメモリビットマップのすべてのビットを設定する方法です。これには理由がありますが、これについてはすぐに説明します。

```
void pmmngr_init (size_t memSize, physical_addr
bitmap) {
    _pmmngr_memory_size      = memSizeです。
    _pmmngr_memory_map      = (uint32_t*)のビットマップです。
    _pmmngr_max_blocks      = (pmmngr_get_memory_size()*1024) / PMMNGR_BLOCK_SIZE。
    _pmmngr_used_blocks     = pmmngr_get_block_count ()を使用しています。

    //!デフォルトでは、すべてのメモリが使用されています
    memset (_pmmngr_memory_map, 0xf, pmmngr_get_block_count() / PMMNGR_BLOCKS_PER_BYTE);
}
```

pmmngr_init_region () - 使用するメモリの領域を初期化する

メモリマップを覚えていますか?私たちは、メモリのどの領域を使っても安全なのかを知りませんが、カーネルだけは知っています。そのため、デフォルトではすべてのメモリが使用されています。カーネルは、カーネルからメモリマップを取得し、このルーチンを使用して、私たちが使用できるメモリの利用可能な領域を初期化します。

このルーチンは非常にシンプルです。どのくらいのメモリブロックを設定するかを調べ、メモリビットマップの適切なビットをクリアしてループするだけです。これにより、アロケーション・ルーチンがこれらの空きメモリ・エリアを再び使用できるようになります。

```
void pmmngr_init_region (physical_addr base, size_t
size) {
    int align = base / PMMNGR_BLOCK_SIZE;
    int blocks = size / PMMNGR_BLOCK_SIZE;

    for (; blocks>0; blocks--) {
        mmap_unset (align++);
        _pmmngr_used_blocks--;
    }

    mmap_set (0); //最初のブロックは常にセットされる。これにより、すべてのブロックが0になることはありません。
}
```

最後の `mmap_set()` の呼び出しに注目してください。今回の PMM では、最初のメモリブロックのブロックが常にセットされます。これにより、PMM がアロケーショナーに対して NULL ポインタを返すことができるようになります。また、最初の 64KB のメモリ内で定義されたデータ構造は、IVT (Interrupt Vector Table) や BDA (Bios Data Area) を含め、上書きされたり触られたりしないようになっています。

pmmngr_deinit_region () - メモリの領域を初期化して使用する

このルーチンは上記のルーチンに似ていますが、ビットをクリアする代わりにビットをセットします。ビットが 2 進数の 1 になるので、ビットが表す 4KB のメモリブロックは実質的に予約済みに設定され、このルーチンが呼ばれてもメモリのその領域が触れられることはありません。

わーい。これで、~~void*の使用領域を初期化し非初期化する办法addr~~ PMM を初期化する方法ができたので、次はブロックの割り当て・解除に取り組みます。

pmmngr_alloc_block () および pmmngr_alloc_blocks () - 物理メモリの単一ブロックを割り当てます。

`int blocks = size / PMMNGR_BLOCK_SIZE;`
メモリブロックを確保するのはとても簡単です。すべての物理メモリはすでに存在しているので、必要なのは、空きメモリブロックへのポインタを返すことです。`mmap_first_free()` の検索を使用することで、空きメモリブロックを見つけることができます。また、`mmap_set` を呼び出して `mmap_first_free ()` が返すアドレスと同じフレームを設定していることにも注目してください。これは、割り当てられたばかりのメモリブロックが現在「使用中」であることを示すものです。このルーチンは、割り当てられたばかりの 4KB の物理メモリに `void*` を返します。

```

}
void*pmmngr_alloc_block () {...
    if (pmmngr_get_free_block_count() <=
        0) return 0; //out of
        memory

    int frame = mmap_first_free ()です。
    if (frame == -
        1) //アウトオブメモ
        0を返す。 リー
    mmap_set (フレーム) です。

    physical_addr addr = frame *
    PMMNGR_BLOCK_SIZE;
    _mnmngr_used_blocks++です。
}

(void*)addrを返します。

```

PMM には、もう一つの割り当てルーチン `pmmngr_alloc_blocks()` があります。このルーチンは上記のものとはほぼ同じなので、スペースの関係上、このチュートリアルでは割愛することにしました。`pmmngr_alloc_blocks()` は、単一のブロックではなく、連続したブロックを割り当てる方法を提供します。

pmmngr_free_block () および pmmngr_free_blocks () - 物理メモリのブロックを解放する

さて、これで物理メモリのブロックを割り当てる方法ができました。次に、メモリ不足にならないように、これらのメモリブロックを解放する方法が必要です。これは簡単すぎます。

`pmmngr_free_blocks()` はほとんど同じように動作しますが、`pmmngr_alloc_blocks()` と併用することで、単一のブロックではなく連続したブロックを解放することができます。

デモ

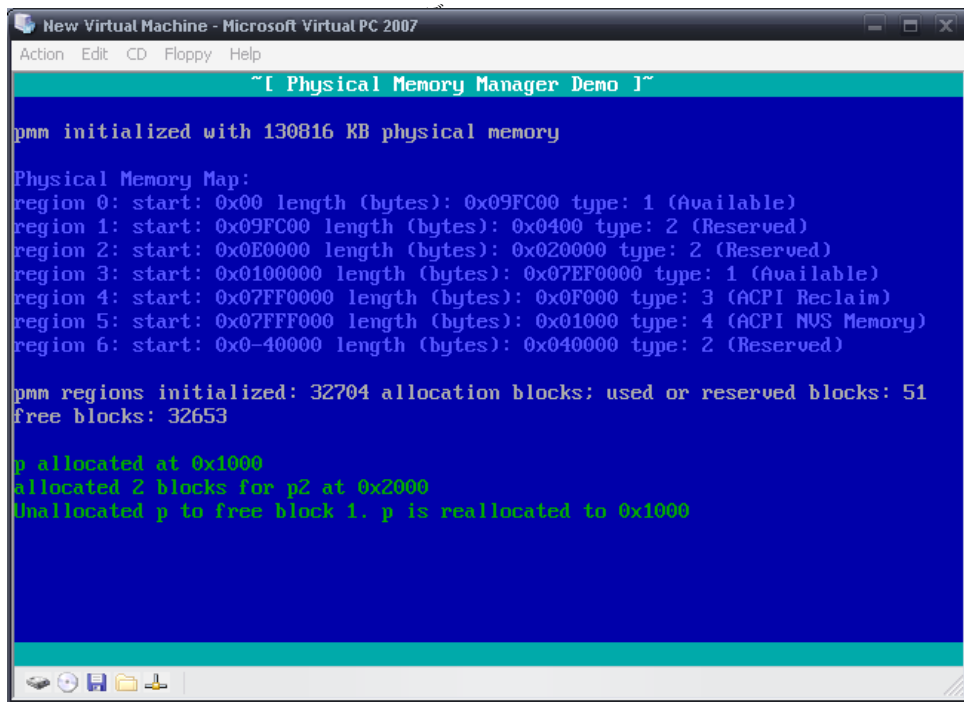
```

physical_addr addr =
    (physical_addr)p; int frame = addr /
    PMMNGR_BLOCK_SIZE;

    mmap_unset (frame)です。

    _mnmngr_used_blocks--。
}

```

VirtualPC上で動作するPhysical Memory Manager

デモダウンロード(MSVC++)

少しスペースを効かせることにした。**Bochs**のメモリマップはつまらない。)このデモは、様々なマシンで実行して、異なるコンピュータシステムが物理メモリの領域をどのようにマッピングするかを見ることができるとい点で、非常に素晴らしいものです。このデモは、すべての光化学レベルで動作するはずですが、

このデモでは、ブートローダから渡されたマルチブート情報構造体を使って、物理メモリのサイズを取得しています。私のVirtualPCは130MBのメモリを使用するように設定されているので、かなりうまく検出できたと言えるでしょう :)

このデモでは多くのことが行われています。第二段階のブートローダのいくつかの更新は、上で見た方法でメモリを検出するために使用され、その情報を上で見たマルチブート情報構造体を使用してカーネルに渡します。

割り当てと解放を試してみて、コードの動作を確認してください。割り当てルーチンのいずれかがヌル (0) を返した場合は、空きブロックがもうないことを示しています。メモリ不足の場合は、メモリを解放するか、代わりにスタックまたはグローバルにオブジェクトを割り当ててみてください。

ここで重要なことがあります。すべての割り当てが4kの境界にアラインされていることに注目してください。これは、次のチュートリアルでページや仮想メモリの話をするときに、とても重要な特徴です。

結論

このチュートリアルは悪くなかったと思います。

最初に、物理メモリ自体を見て、物理メモリとは何か、どのように機能するのか、物理アドレス空間とアドレッシングモードを理解しました。また、BIOSからメモリ情報を取得し、それをカーネルに与える方法や、物理メモリマネージャの開発についても見てきました。

これで、物理的なメモリブロックを割り当てたり解放したりする方法ができました。これは素晴らしいことですが、まだ問題があります。つまり、ファイルやプログラムを読み込む場合、物理メモリマネージャを使って、そのファイルやプログラムに十分な大きさのメモリ領域を割り当てればよいのです。しかし.....もし、十分な大きさの領域がなかったら？これは、ロードされるプログラムは、カーネルによってロードされる特定のアドレスにリンクされなければならないということでもあります。

そこで登場するのが、仮想メモリとページングです。次のチュートリアルでは、ページングと仮想メモリについて見ていきます。このチュートリアルでは、4GBのアドレス空間全体をマッピングして制御する方法を学びます。また、仮想アドレスとは何か、そしてそれを使って上記の問題を解決する方法などを見ていきます。それでは、会場でお会いしましょう。

次の機会まで。

マイク

BrokenThorn Entertainment 社。現在、DoE と Neptune Operating System を開発中です。質問

やコメントはありますか？お気軽にお問い合わせください。

あなたも記事の改善に貢献したいと思いませんか？もしそうなら、ぜひ私に教えてください。



第16章

ホーム

第18





オペレーティングシステム開発シリーズ

オペレーティングシステム開発 - 仮想メモリ

by Mike, 2008

このシリーズは、オペレーティングシステムの開発を一から実演し、教えることを目的としています。

はじめに

おかえりなさい。もうチュートリアル18回目を迎えたなんて信じられないよ。ほらね。OS開発も悪くないでしょ？)

前回のチュートリアルでは、物理的なメモリ管理について説明し、実際に動作する物理的なメモリマネージャの開発も行いました。このチュートリアルでは、ページングと仮想メモリを導入することで、新たなレベルに進みます。このチュートリアルでは、ページングと仮想メモリを紹介し、プログラムに完全な仮想アドレス空間を模倣する方法と、仮想メモリを管理する方法を学びます。

この章のリストを紹介します。

- 仮想メモリ
- メモリマネージメントユニット (MMU)
- TLB (Translation Lookaside Buffer)
- PAE and PSE
- ページングの方法
- ページとページ・フォー
- ルト ページ・テーブル
- ページディレクトリテーブル
- ルによるページングの実現

...他にもいろいろあります。

このチュートリアルでは、前章で開発した物理メモリマネージャをベースにしています。これがメモリ管理の最後の章になるかもしれません。

それでは、早速始めてみましょう。

仮想メモリの概念

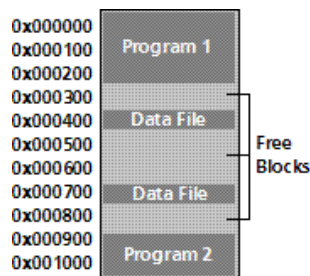
仮想化の必要性

なぜ「仮想メモリ」を気にしなければならないのか、不思議に思われるかもしれません。結局のところ、私たちはすでにメモリを管理する素敵で効果的な方法を持っているのではないのでしょうか？まあ、そんなところです。物理メモリマネージャは、メモリブロックをうまく管理しますが、それがすべてです。これだけでは、かなり意味がないと思いませんか？

仮想メモリとその必要性を理解するためには、非常に重要な概念がたくさんあります。

今のところ、私たちが持っているのは、物理的なメモリを直接または間接的に操作する方法だけです。この方法には、皆さんもご存知の（あるいは経験のある）大きな問題がたくさんあります。先ほど見たのは、存在しないメモリブロックにアクセスする場合でした。プログラムとデータの両方がメモリ上にあることを考えると、プログラムがお互いのメモリ空間にアクセスしたり、知らないうちに自分や他のプログラムを破壊したり上書きしたりすることも可能なのです。何しろ、メモリ保護がないのだから。

また、ファイルやプログラムをメモリの連続した領域に読み込むことができない場合もあります。これがフラグメント（断片化）です。例えば、2つのプログラムがロードされているとします。1つは0x0で、もう1つは0x900です。これらのプログラムは両方ともファイルのロードを要求していたので、データファイルをロードします。



ここで何が起きているのかを見てみましょう。これらすべてのプログラムとファイルの間には、未使用のメモリがたくさんあります。では、さらに大きなファイルを追加して、上記に収まらなくなったらどうなるでしょう？ここで、現在の方式では大きな問題が発生します。現在実行中のプログラムや読み込まれているファイルを破壊してしまうため、特定の方法で直接メモリを操作することはできません。

このように、物理メモリを扱う際には多くの問題が発生します。オペレーティングシステムがシングルタスク（一度に1つのリング0プログラムしか実行しない）であれば、これで問題ないかもしれません。もっと複雑なものになると、システム内でのメモリの動きをもっとコン

このような細かいことを気にしなくてもいいようになっています^ズ。ここで私が言いたいことは、仮想化の出番だということがお分かりいただけると思います。早速見てみましょう。

バーチャルメモリー

概念

仮想メモリとは何かを理解するのは、少し難しいことです。仮想メモリとは、ハードウェアとソフトウェアの両方で採用されている特別なメモリアドレススキームです。隣接していない物理メモリを、あたかも隣接したメモリのように扱うことができます。

「メモリアドレス方式」と言ったことに注目してください。これはどういうことかという、仮想メモリでは、**Memory Address**が何を参照するかをコントロールできるということです。

仮想アドレス空間 (VAS)

仮想アドレス空間とは、プログラムのアドレス空間のことです。ここで注意しなければならないのは、これは物理的なメモリとは関係ないということです。これは、各プログラムがそれぞれ独立したアドレス空間を持つことを意味します。これにより、あるプログラムが別のプログラムにアクセスできないようにします。

VASは仮想的なものであり、物理的なメモリを直接使用するものではないため、ディスクドライブなどの他のソースをメモリのように使用することができます。つまり、システムに物理的に搭載されている以上の「メモリ」を使用することができるのです。

これにより、「メモリが足りない」という問題が修正されます。

また、各プログラムが独自のVASを使用しているため、各プログラムは常にベース0x0000:0000から始まるようにすることができます。これにより、前述の再配置の問題や、メモリの断片化が解消され、各プログラムに連続した物理ブロックを割り当てる心配がなくなります。

仮想アドレスは、**MMU**を介してカーネルによってマッピングされます。これについては、もう少し後に説明します。

メモリマネジメントユニット (MMU)

MMU (Memory Management Unit) (別名: **PMMU (Paged Memory Management Unit)**) は、マイクロプロセッサとメモリコントローラの間に設置される。メモリコントローラの主な機能は、メモリアドレスを物理的なメモリロケーションに変換することですが、**MMU**の目的は、仮想メモリアドレスをメモリコントローラが使用するメモリアドレスに変換することです。

つまり、ページングが可能になると、すべてのメモリ参照が最初に**MMU**を経由することになるのです。

TLB (Translation Lookaside Buffer) について

これは、仮想アドレス変換の速度を向上させるためにプロセッサ内に保存されるキャッシュです。通常は**CAM (Content-addressable memory)**の一種で、変換する仮想アドレスを検索キーとし、その結果を物理的なフレームアドレスとしています。そのアドレスがTLBに存在しない場合 (**TLBミス**)、**MMU**はページテーブルを検索してアドレスを見つけます。TLBに見つかった場合、それは**TLBヒット**となります。TLBミスの際に、ページがページテーブル内で見つからなかったり、無効だったりした場合は、プロセッサから**ページフォルト**例外が発生します。

TLBは、RAMではなくキャッシュに格納されたページのテーブルと考えるとよいでしょう。

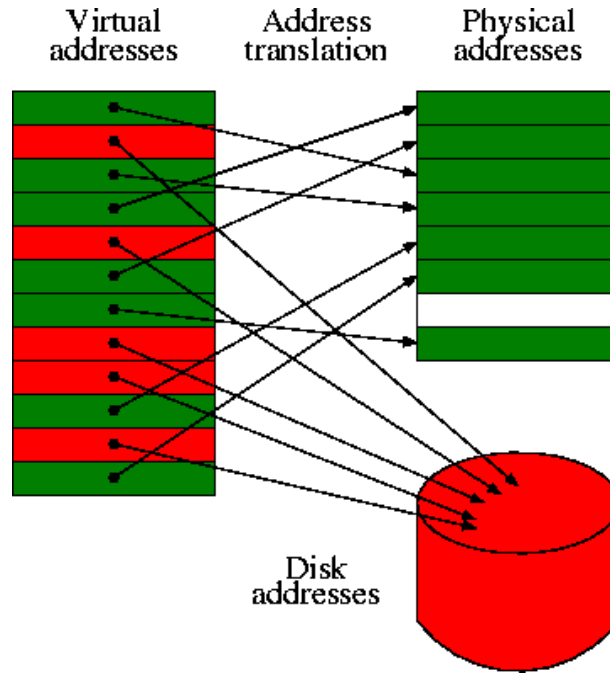
これは重要なことです。ページは**ページテーブル**に格納されています。このページテーブルを設定することで、物理アドレスが仮想アドレスにどのように変換されるかを記述します。言い換えれば**TLB**は、**私たちが設定したページテーブルを使って、仮想アドレスを物理アドレスに変換しています**。そう、どの仮想アドレスが何に対応するかを設定するのです。この方法については後ほどご紹介します。心配しなくても、そんなに悪いことはありません。)

ページングされた仮想メモリ

また、仮想メモリは、システムに実際に搭載されているよりも多くのメモリを間接的に使用方法でもあります。一般的な方法として、ハードドライブや**スワップパーティション**に保存されている**ページファイル**を使用する方法があります。

仮想メモリは、ハードウェアレベルで処理されるため、動作させるためにはハードウェアデバイスコントローラを介してマッピングする必要があります。これは通常、後述する**MMU**を通じて行われます。

仮想メモリの使用例として、実際に見てみましょう。



ここで何が起きているのかに注目してください。仮想アドレス内の各メモリブロックは直線的ですが、各メモリブロックは、実際の物理的なRAM内の位置か、ハードディスクなどの別のデバイスにマッピングされています。このブロックは、必要に応じてこれらのデバイス間でスワップされます。これは遅く見えるかもしれませんが、MMUのおかげで非常に高速です。

覚えておいてください。各プログラムは、上図のような仮想アドレス空間を持ちます。各アドレス空間は0x0000:00000から始まる直線的なものであるため、メモリの断片化やプログラムの再配置に関する問題のほとんどが解決されます。

また、仮想メモリはメモリブロックを使用するデバイスが異なるため、システム内のメモリ量以上の管理を容易に行うことができます。例えば、システムメモリが足りなくなった場合、代わりにハードディスクにブロックを割り当てることができます。メモリが足りなくなったら、必要に応じてこのページファイルを増やすか、警告やエラーのメッセージを表示することができます。

各メモリ「ブロック」は「ページ」と呼ばれ、通常4096バイトの大きさです。ページについては後ほど説明します。

さて、ページとはメモリブロックのことです。このメモリブロックは、メモリ内の場所にマッピングされる場合と、ハードディスクなどの他のデバイスの場所にマッピングされる場合があります。これはマッピングされていないページです。ソフトウェアがマッピングされていないページにアクセスした場合（そのページは現在メモリ内にありません）、何らかの方法でそれを読み込む必要があります。これを行うのが、ページフォルトハンドラです。

すべては後で説明しますので、難しいと思われるかもしれませんがご心配なく。）

ここでは一般的なページングの話をしているので、ページングで使用される可能性のあるいくつかの拡張機能を見てみるのが良いと思います。早速見てみましょう。

PAEとPSE

フィジカル・アドレス・エクステンション (PAE)

PAEは、x86マイクロプロセッサの機能の一つで、32ビットシステムが最大64GBの物理メモリにアクセスできるようにするものです。PAEをサポートするマザーボードでは、この機能を実現するために36行のアドレスバスを使用しています。PAEを有効にした場合のページングサポート（cr4レジスタのビット5）は、これまで見てきたものとは少し異なります。しかし、このチュートリアルがさらに複雑にならないように、今はこの点については触れません。しかし、興味のある方はぜひ調べてみてください。）

PSE (Page Size Extension) について

PSEとは、x86マイクロプロセッサの機能の一つで、4KB以上のページサイズを可能にするものです。これにより、x86アーキテクチャでは、4KBのページに加えて4MBのページサイズ（「巨大ページ」または「大規模ページ」とも呼ばれる）をサポートすることができます。

ページングの世界

狂気の沙汰を始めよう！)

はじめに

わーい！わーいページングという素晴らしく、ひねくれた世界へようこそ。これまでに説明してきた基本的な概念により、ページングや仮想メモリがどのようなものであるかを十分に理解いただけたと思います。これは素晴らしいスタートだと思いませんか？

2021/11/15 13:09

オペレーティングシステム開発シリーズ

いいですね。でも、実際にそれをどうやって実現するのでしょうか？~~か~~ **x86**アーキテクチャでは、ページングはどのように機能するのでしょうか？それを見てみましょう。

ページ

ページ（メモリページまたは**仮想ページ**とも呼ばれる）は、固定長のメモリブロックです。このメモリブロックは、物理メモリに常駐することができます。このように考えてください。ページには、メモリブロックとその位置が記述されています。これにより、そのメモリブロックがどこにあるかを「マッピング」または「検索」することができます。ページのマッピングとページングの実装方法については、もう少し後に説明します。）

i86アーキテクチャでは、まさにこのための特別なフォーマットを使用しています。これにより、1つのページと、そのページが現在どこにあるのかを追跡することができます。それでは見てみましょう。

ページテーブルエントリ（PTE）

ページテーブルのエントリは、ページを表すものです。ページテーブルについては、もう少し後になってから説明しますので、あまり気にしないでください。しかし、テーブルのエントリがどのようなものかを見ておく必要があります。x86アーキテクチャでは、ページを扱うための特定のビットフォーマットが定義されているので、それを見てみましょう。

- **ビット0 (P)**。現在のフラグ
 - 0: ページがメモリにない
 - 1: ページが存在する（メモリ内）
- **ビット1 (R/W)**。リード/ラ
 - イットフラグ 0: ページはリードのみ
 - 1: ページが書き込み可能
- **Bit 2 (U/S): User mode/Supervisor mode**
 - flag 0: ページはカーネル（スーパーバイザー）モード
 - 1: ページはユーザーモード。スーパーバイザーページの読み書きができない
- **ビット3-4 (RSVD)**。インテルによる予約済み
- **ビット5 (A)**。アクセスフラグ。プロセッ
 - サによって設定される 0: ページはアクセスされていない
 - 1: ページにアクセスした
- **ビット6 (D)**。ダーティ フラグ。プロセ
 - ッサによって設定される 0: ページが書き込まれていない 1: ページが書き込まれている
- **ビット7-8 (RSVD)**。予約
- **ビット9-11 (AVAIL)**。使用可能
- **ビット12~31 (FRAME)**。フレームアドレス

CoolDOS!これで終わりですか？まあね。難しいとは言っ

ていませんよ;)

ここで最も重要なのは、**フレームアドレス**です。フレームアドレスは、**そのページが管理する4KBの物理メモリの位置を表します**。これはページングを理解する上で**重要なこと**ですが、その理由を今から説明するのは難しいです。とりあえず、**1つ1つのページが1つのメモリブロックを管理していることを覚えておいてください**。ページが存在するということは、**物理メモリ上の4KBの物理アドレス空間を管理している**ということです。

ダーティフラグとアクセスフラグは、ソフトウェアではなくプロセッサが設定します。プロセッサがどのビットを設定すべきかをどのようにして知るのか、つまりメモリ内のどこに位置するのか、ということが気になるかもしれません。これについては後ほど説明します。このフラグを設定することで、ソフトウェアやエグゼクティブは、あるページがアクセスされたかどうかをテストすることができるのです。

現在フラグは重要なものです。この1ビットで、あるページが現在物理メモリにあるかどうかを判断します。物理メモリ内にある場合、フレームアドレスはそのページがある場所の32ビットリニアアドレスとなります。物理メモリ内にない場合、そのページはハードディスクなどの別の場所に存在しているはずですが。

現在フラグが設定されていない場合、プロセッサは構造体の残りのビットを無視します。これにより、残りのビットを何かの目的に使用することができます。例えば、ディスク上のページの位置などです。これにより、ページフォルトハンドラが呼び出されたときに、ディスク上のページを探し出し、必要に応じてページをメモリにスワップすることができます。

簡単な例を挙げてみましょう。例えば、このページは物理位置1MB（0x100000）から始まる4KBのアドレス空間を管理したいとします。これは言い換えれば、**このページが1MBのアドレスに「マッピング」されていることを意味します**。

このページを作成するには、ページの12-31ビット（フレームアドレス）に0x100000を設定し、現在のビットを設定するだけです。そうすれば、ページは1MBにマッピングされます(笑)。例えば

```
%define          PRIV          3

mov              ebx, 0x100000 |          PRIV; このページは1MBにマッピン
ab1              グされている
e
```

0x100000が4KBアラインされていることに注目してください。これを3（11バイナリ）とORして、最初の2ビットを設定します。上の表を見ると、「現在」と「読み取り/書き込み」のフラグが設定され、このページが「現在」になっていることがわかります（物理メモリ上に

あることを意味します。物理アドレス0x100000からマッピングされているので、これは事実です)、そして書き込み可能になります。

これでおしまいです。この例は、次のいくつかのセクションでさらに拡大され、すべてがどのように組み合わされているかがわかるようになっていきます。

また、PTEは特別なものではなく、単なる32ビットのデータであることに注意してください。PTEが特別なのは、PTEがどのようにしてを使用しています。それはもう少し後に見てみましょう...

pte.h, pte.cpp - ページテーブルのエントリとページの抽象化

このデモでは、ページテーブルエントリの個々のプロパティを設定・取得するコードを、これら2つのファイルの中にすべて隠しています。これらが行うのは、上のリストで見た32ビットパターンからビットとフレームアドレスを設定・取得することだけです。このインターフェースには多少のオーバーヘッドがありますが、読みやすさが大幅に向上し、作業がしやすくなります。

まず最初に行うことは、ページテーブルエントリで使用されるビットパターンの抽象化です。これは簡単すぎます。

```
enum PAGE_PTE_FLAGS {
    i86_pte_present          = 1,          //00000000000000000000000000000001
    i86_pte_writable         = 2,          //00000000000000000000000000000010
    I86_PTE_USER             = 4,          //00000000000000000000000000000100
    i86_pte_writethrough     = 8,          //000000000000000000000000000001000
    i86_pte_not_cacheable    = 0x10,      //0000000000000000000000000000010000
    i86_pte_accessed         = 0x20,      //0000000000000000000000000000010000
    I86_PTE_DIRTY            = 0x40,      //00000000000000000000000000000100000
    I86_PTE_PAT              = 0x80,      //000000000000000000000000010000000
    i86_pte_cpu_global       = 0x100,     //000000000000000000000000010000000
    (i86_pte_cpu_global)
    i86_pte_lv4_global       = 0x200,     //0000000000000000000000000100000000
    I86_PTE_FRAME            = 0x7FFFF000 //1111111111111111111111110000000000
};
```

これが上のリストで見たビットフォーマットと一致することに注目してください。私たちが欲しいのは、これらのプロパティ（つまり、ビット）の設定と取得をインターフェースの背後で抽象化する方法です。

そのためには、まず、ページテーブルのエントリを格納するためのデータ型を抽象化します。今回の例では、単純なuint32_tです。

簡単でいい。これは、これらのビットを設定したり取得したりするために使用されるインターフェース ルーチンです。このルーチンはpt_entry内の個々のビットを設定したり取得したりするだけなので、その実装については見たくありません。その代わりに、インターフェースに注目したいと思います。

```
extern void pt_entry_add_attrib (pt_entry* e, uint32_t attrib); extern
void pt_entry_del_attrib (pt_entry* e, uint32_t attrib); extern
void pt_entry_set_frame (pt_entry*, physical_addr);
extern bool pt_entry_is_present (pt_entry e) です。
extern bool pt_entry_is_writable (pt_entry e); extern
physical_addr pt_entry_pfn
(pt_entry e);
```

pt_entry_add_attrib() は、pt_entry 内の 1 つのビットを設定します。マスク (I86_PTE_PRESENT ビットマスクのようなもの) を渡して設定します。 **pt_entry_del_attrib()** は同じことを行いますが、ビットをクリアします。

pt_entry_set_frame() は、フレームアドレスをマスクアウト (I86_PTE_FRAME マスク) して、それに自分のフレームアドレスを設定します。**pt_entry_pfn()** は、このアドレスを返します。

これらのルーチンは特別なものではありません。必要であれば、ビットマスクや（必要であれば）ビットフィールドを使って、これらの属性を簡単に手動で設定・取得することができます。個人的には、この設定の方がはるかに作業しやすいと感じていますが。）

この設定では、1つのページを追跡することができるので、これは素晴らしいことだと思います。しかし、一般的なシステムでは多くのページを持つ必要があるため、これだけでは意味がありません。そこで、ページテーブルの出番です。

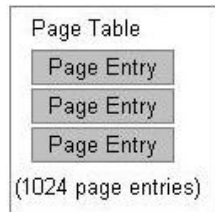
ページテーブル

ページテーブル.....うーん、どこかで聞いたような言葉だな。*looks one line up*.ああ、そうか ;)

ページテーブルとは...そう、ページの表のことです。（ページテーブルは、物理アドレスと仮想アドレスの間でページがどのようにマッピングされているかを追跡することができます。このテーブルの各ページエントリは、前のセクションで示したフォーマットに従います。つまり、**ページテーブル**とは、**ページテーブルエントリ (PTE)** の配列のことです。

ページテーブルは非常にシンプルな構造ですが、非常に重要な目的を持っています。ページテーブルには、含まれているすべてのページのリストと、それらがどのようにマッピングされているかが記載されています。マッピング」とは、仮想アドレスが物理的なフレームアドレスにどのように「マッピング」されるかを意味します。また、ページテーブルは、ページの管理、ページが存在する環境、ページの保存方法、さらにはページがどのプロセスに属しているかまでも管理します（これは、ページの**AVAIL**ビットを使用して設定できます。これは必要ないかもしれませんが、システムの実装によります）。

ちょっと立ち止まって考えてみましょう。**ページは、4KBの物理アドレス空間を管理していることを覚えていますか？**各ページが4KBの物理メモリを「管理」しているので、1024ページを合わせると1024*4KB=4MBの仮想メモリを管理していることになります。では、どのように設定されているか見てみましょう。



```
struct page_table {
    page_entry m_entries[1024];
};
```

これがページテーブルの例です。ページテーブルは、**1024個**のページエントリの配列に過ぎないことに注目してください。各ページが**4KB**の物理メモリを管理していることを考えると、この小さなテーブルを独自の**仮想アドレス空間**に変えることができます。どうすればいいのでしょうか？簡単です。それは、**仮想アドレス**のフォーマットを決めることです。

例を挙げてみましょう。例えば、次のような新しいバーチャルアドレスのフォーマットを設計したとします。

```
AAAAAAAAAA     BBBBBBBBBBBB
ページテーブルインデックスページ内のオフセット
```

これが仮想アドレスのフォーマットです。つまり、ページングを有効にすると、すべてのメモリアドレスが上記の形式になります。例えば、次のような命令があったとします。

```
mov     ecx,
    [0xc0000].
e
```

ここでは、**0xc0000**が**仮想アドレス**のように扱われます。これを分解してみましょう。

```
11000000      000000000000 ; 0xc0000のバイナリ形式
AAAAAAAAAA     BBBBBBBBBBBB
ページテーブルインデックスページ内のオフセット
```

現在行っているのは、**アドレス変換**の一例です。実際にこの仮想アドレスを変換して、それがどの物理的な場所を指しているかを確認しています。ページテーブルのインデックスは、**11000000b = 192**です。これは、ページテーブルの中のページエントリです。これで、このページが管理する**4KB**のベース物理アドレスを得ることができます。このページが存在する（**Pages present**フラグが設定されている）場合、必要なのはページの**フレームアドレス**にアクセスしてメモリにアクセスすることだけです。このページが存在しない場合は、ページフォルトを生成します--ページデータはディスクのどこかにあるかもしれません。ページフォルトハンドラは、ページの**4KB**のデータをどこかのメモリにコピーし、ページを**現在**に設定し、その**フレームアドレス**を更新して、この新しい**4KB**の物理メモリブロックを指すようにします。

はいはい、わかっていますよ。偽の "バーチャル・アドレス" を作成するこの小さな例は、馬鹿げているように見えるかもしれませんが、何だと思いませんか？**実際にはこのようにして作られます**。実際のバーチャルアドレスのフォーマットは、**2つの**セクションではなく**3つの**セクションがあるという点で、少し複雑になっています。

ここまでくると、すべてがどのように組み合わされているのか、そしてページテーブルの重要性がわかってきたのではないのでしょうか。

ページサイズ

ページサイズが小さいシステムでは、ページサイズが大きいシステムよりも多くのページが必要になります。テーブルはすべてのページを追跡するので、ページサイズが小さいシステムは、追跡するページが多いため、より大きなページテーブルが必要になります。簡単でしょうか？

i86アーキテクチャでは、**4MB (PAE (Page Address Extension))**を使用する場合は**2MBページ** および**4KBサイズ**の

ページをサポートしています。注意すべき点は以下の通りです。ページサイズがページテーブルのサイズに影響を与える可

能性があることに注意してください。

ページディレクトリテーブル (PDT) について

よし...。もうすぐ完成ですね。ページテーブルは、非常に強力な構造です。前回の仮想アドレスの例を覚えていますか？私は、各仮想アドレスが**2つの**部分から構成される仮想アドレスシステムの例を挙げました。ページテーブルのエントリと、そのページへのオフセットです。

x86アーキテクチャでは、仮想アドレスフォーマットは**2つの**セクションではなく、実際には**3つの**セクションを使用しています。**ページディレクトリテーブル**のエントリ番号、ページテーブルのインデックス、そしてそのページのオフセットです。

ページディレクトリテーブルは、**ページディレクトリエントリ**の配列に他なりません。わかっている、わかっている。最後の一文は、ただ無駄で情報量の少ないものだったのでしょうか;))

さて、まずはページのディレクトリエントリを見てみましょう。次に、ディレクトリテーブルを見ていきますが、その中には...

PDE (ページディレクトリエントリ) について

ページディレクトリエントリは、単一のページテーブルを管理する方法を提供するのに役立ちます。ページディレクトリエントリは、ページ

テーブルのアドレスを含んでいるだけでなく、ページテーブルを管理するためのプロパティを提供しています。次のセクションでは、これらすべてがどのように組み合わせられるかを説明しますので、まだ理解できなくても心配しないでください。

ページディレクトリテーブルは、ページテーブルの構造と非常に似ています。PDEは1024個のエントリの配列で、エントリは特定のビットフォーマットに従います。ページディレクトリエントリ(PDE)のフォーマットの良いところは、ページテーブルエントリ(PTE)とほぼ同じフォーマットに従っていることです(実際には交換可能です)。ただ、少しだけ細かい点があります(シャレですが;)。

ここでは、ページディレクトリエントリのフォーマットを紹介します。

- **ビット0 (P)**。現在のフラグ
 - 0: ページがメモリにない
 - 1: ページが存在する (メモリ内)
- **ビット1 (R/W)**。リード/ラ
 - イットフラグ 0: ページはリードのみ
 - 1: ページが書き込み可能
- **Bit 2 (U/S): User mode/Supervisor mode**
 - flag 0: ページはカーネル (スーパーバイザー) モード
 - 1: ページはユーザーモード。スーパーバイザーページの読み書きができない
- **ビット3 (PWT)** : ライトスルーフラグ
 - 0: ライトバックキャッシングが有効
 - 1: ライトスルーキャッシングが有効
- **ビット4 (PCD)** : キャッシュディセーブル
 - 0: ページテーブルをキャッシュし
 - ない 1: ページテーブルをキャッシュする
- **ビット5 (A)**。アクセスフラグ。プロセッ
 - サによって設定される 0: ページはアクセスされていない
 - 1: ページにアクセスした
- **ビット6 (D)** です。インテルによる予約
- **ビット7 (PS)** です。
 - ページサイズ0: 4KBページ
 - 1: 4 MBページ
- **ビット8 (G)**。グローバルページ (無視)
- **ビット9-11 (AVAIL)**。使用可能
- **ビット12~31 (FRAME)**。ページテーブルベースアドレス

ここにいるメンバーの多くは、先日ご紹介したページテーブルエントリ (PTE) のリストに見覚えがあるはずです。

Present、Read/Write、AccessのフラグはPTEの時と同じですが、ページではなくページテーブルに適用されます。

ページサイズは、ページテーブル内のページが**4KB**なのか**4MB**なのかを決定します。ページ

テーブルのベースアドレスビットには、ページテーブルの4Kアラインメントアドレスが含ま

れています。 **pde.h**および**pde.cpp** - ページディレクトリエントリの抽象化

PTEで行ったことと同様に、PDEを抽象化するためのインターフェースを作成しました。

```
enum PAGE_PDE_FLAGS
{
    i86_pde_present          = 1,           //00000000000000000000000000000001
    i86_pde_writable         = 2,           //00000000000000000000000000000010
    I86_PDE_USER             = 4,           //00000000000000000000000000000100
    I86_PDE_PWT              = 8,           //000000000000000000000000000001000
    I86_PDE_PCD              = 0x10,        //0000000000000000000000000000010000
    i86_pde_accessed         = 0x20,        //00000000000000000000000000000100000
    I86_PDE_DIRTY            = 0x40,        //000000000000000000000000000001000000
    I86_PDE_4MB              = 0x80,        //0000000000000000000000000000010000000
    i86_pde_cpu_global       = 0x100,       //00000000000000000000000000000100000000
    (i86_pde_cpu_global)
    i86_pde_lv4_global       = 0x200,       //0000000000000000000000000000010000000000
    I86_PDE_FRAME           = 0x7FFFF000   //1111111111111111111111111000000000000000
};

// ! ページの演出エントリ
typedef uint32_t
pd_entry;
```

難しいことはありません。ページディレクトリエントリを表すために、新しい型 **pd_entry** を使用します。また、PTE インターフェースでは、ページディレクトリエントリ内のビットを設定および取得するための優れた方法を提供するために使用される小さなルーチンセットを提供しています。

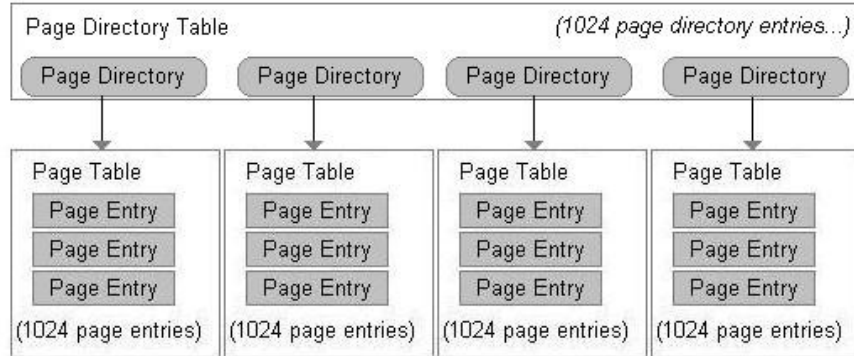
```
extern void pd_entry_add_attrib (pd_entry* e, uint32_t attrib); extern
void pd_entry_del_attrib (pd_entry* e, uint32_t attrib); extern
void pd_entry_set_frame (pd_entry*, physical_addr);
extern bool pd_entry_is_present (pd_entry e) です。
extern bool pd_entry_is_user (pd_entry) です。
extern bool pd_entry_is_4mb (pd_entry) です。
```

ページディレクトリテーブルについて

ページディレクトリテーブルは、**1024個**のページテーブルの配列のようなものです。各ページテーブルが**4MB**の仮想アドレス空間を管理することを覚えていますか？さて...。**1024個**のページテーブルを組み合わせれば、**4GB**の仮想アドレスを管理することができます。すごいでしょ？

さて、それでは少し複雑になってしまいますが、それほどでもありません。**ページディレクトリテーブルは、1024個のページディレクトリエントリの配列で、上記のフォーマットに従っています。エントリのフォーマットを見て、ページテーブルベースのアドレスビットに注目してください。**これは、このディレクトリエントリが管理するページテーブルのアドレスです。

視覚的に見た方がわかりやすいかもしれませんので、ご紹介します。



ここで起きていることに注目してください。各ページディレクトリエントリは、ページテーブルを指しています。各ページが**4KB**の物理メモリ（つまり仮想メモリ）を管理していることを覚えていますか？また、ページテーブルは、**1024ページ**の配列でしかないことを覚えていませんか？

1024*4kb=4MBです。つまり、各ページテーブルが**4MB**のアドレス空間を管理していることになります。

各ページディレクトリエントリは、各ページテーブルをより簡単に管理する方法を提供します。完全なページディレクトリテーブルは、**1024個**のディレクトリエントリの配列であり、各エントリがそれぞれのテーブルを管理するため、実質的に**1024個**のページテーブルを持つことになります。先ほどの計算で、各ページテーブルが**4MB**のアドレス空間を管理することがわかりました。つまり、**1024のページテーブル×4MBのサイズ=4GB**の仮想アドレス空間ということになります。

信じられないかもしれませんが...これで全部です。ほら、そんなに難しくないでしょう？次のセクションでは、私たちは再び**x86**の仮想アドレスの**実際の**フォーマットを見て、すべてがどのように連動しているかを知ることができます。

マルチタスクでの使用

ここでちょっとした問題が発生します。ページディレクトリテーブルは、**4GB**のアドレス空間を表していることを覚えていますか？一度に**1つ**のページディレクトリしか持てないのであれば、どうやって複数のプログラムに**4GB**のアドレス空間を与えることができるでしょうか？

できません。ネイティブではありませんが。多くのマルチタスクOSでは、自身が使用する上位**2GB**のアドレス空間を「カーネル空間」、下位**2GB**を「ユーザー空間」としてマッピングしています。ユーザースペースはカーネルスペースに触れることはできません。カーネルのアドレス空間が各プロセスの**4GB**の仮想アドレス空間にマッピングされているので、どのプロセスが実行中であっても、カーネルを使ってカレントページのディレクトリをエラーなく切り替えることができます。これは、カーネルがプロセスのアドレス空間の同じ場所に常に配置されているからです。これにより、スケジューリングも可能になります。詳しくは後述しますが...

仮想メモリの管理

ここまでで、優れた仮想メモリ・マネージャを開発するために必要なことをすべて網羅しました。仮想メモリマネージャは、ページ、ページテーブル、ページディレクトリテーブルを割り当て、管理する方法を提供しなければなりません。これまで、それぞれを個別に見てきましたが、それらがどのように連携するかについては見てきませんでした。

ハイアーハーフケルン

アブストラクト

ハイアーハーフカーネルとは、仮想ベースアドレスが**2GB**以上のカーネルのことです。多くのOSがハイアーハーフカーネルを持っています。例えば、**Windows**カーネルや**Linux**カーネルなどがあります。**Windows**カーネルは**2GB**または**3GB**の仮想アドレスにマッピングされ（/3gbカーネルスイッチを使用するかどうかによる）、**Linux**カーネルは**3GB**の仮想アドレスにマッピングされます。このシリーズでは、**3GB**にマッピングされた上位ハーフカーネルを使用しています。ハイアーハーフカーネルは、仮想アドレス空間に適切にマッピングされなければなりません。これを実現するにはいくつかの方法がありますが、ここではその一部をご紹介します。

なぜ高いハーフカーネルが必要なのか、興味があるかもしれません。カーネルを低い仮想アドレスで実行することは十分可能です。一つの理由は、**v86**タスクと関係があります。**v86**タスクをサポートしたい場合、**v86**タスクはユーザーモードで、リアルモードのアドレス制限(0xffff:0xffff)、つまり約**1MB+64k**のリニアアドレス内ではしか実行できません。また、ユーザーモードのプログラムは、最初の**2GB**（OSによっては**3GB**）で実行するのが一般的です。これは、ソフトウェアが高いメモリ位置にアクセスする必要がないからです。

方法1

1つ目の設計は、ブートローダーに一時的なページディレクトリを設定させることです。これにより、カーネルのベースアドレスを**3GB**に

2021/11/15 13:09

オペレーティングシステム開発シリーズ

することができます。ブートローダは、このベースアドレスに物理アドレス（通常は**1MB**）をマッピングし、カーネルのエントリーポイントを呼び出します。

この方法はうまくいきますが、カーネルが仮想メモリの管理をどのように行うかという問題があります。カーネルは、ブートローダが設定したページディレクトリとテーブルを使って作業するか、新しいページディレクトリを作成して管理するかのどちらかになります。もし

が新しいページディレクトリを作成する場合、カーネルは自分自身をリマップする（1MBの物理アドレスをカーネルのベース仮想アドレスに）か、既存の一時的なページディレクトリを新しいページディレクトリにクローンする必要があります。

現時点では、シリーズではこの方法を採用しています。シリーズのブートローダーは、一時的なページディレクトリを設定し、カーネルを3GBの仮想にマッピングします。その後、VMMの初期化時にカーネルが新しいページディレクトリを作成し、自分自身を再マッピングします。このセットアップ段階では、カーネルは位置に依存しないようにしなければなりません。これは、私たちが社内のOSで使っている方法です。

方法2

また、ブートローダーがカーネルを物理メモリにロードし、ページングを無効にしておくという設計も考えられます。カーネルの仮想ベースアドレスは、本来実行されるべき仮想アドレスになります。例えば、カーネルのベースアドレスは3GBですが、ブートローダーは物理的に1MBのカーネルをロードして実行することができます。

この方法は少し厄介です。ブートローダーがカーネルをどの物理アドレスにロードして実行するかを知る方法が必要で、カーネルは自分自身を実ベースの仮想アドレスにマッピングしなければなりません。これは通常、位置に依存しないコードのカーネル起動時に行われます。これは位置依存のコードでも使用できますが、データへのアクセスや関数の呼び出しの際に、カーネルがアドレスを固定できなければなりません。これは、当社の内製OSで採用している方法です。

方法3

この方法では、Tim RobinsonのGDTトリックを使用します。これにより、カーネルが高いアドレス（ベースアドレス）にロードされていなくても、そのアドレスで実行できるようになります。このトリックは、アドレスラップアラウンドで動作します。例えば、カーネルは1MBの物理アドレスでロードされているが、3GBの仮想アドレスで動作しているように見せたいとします。この場合、 $X + 3GB = 1MB$ が基本となります。もっと詳しく見てみましょう。

GDT記述子のベースアドレスはDWORDであることを覚えておってください。この値が0xffffffffよりも大きくなると、折り返して0に戻ります。 $0xffffffff - 0xc0000000 = 0x3fffffff$ 折り返しまで残り1バイト。このアドレスが物理的な位置（1MB）を指すように、アドレスを追加する必要があります。DWORDが0に戻るまでに0x3fffffffバイト残っていることから、 $0x100000 (1MB) + 0x3fffffff = 0x400fffff + 1 = 0x40100000$ を追加します。

そこで、上記の例を用いて、カーネルが1MBの物理アドレスでロードされているが、実ベースアドレスが3GBの仮想アドレスである場合、ベースコードとデータのセクタが0x40100000である一時的なGDTを作成することができます。プロセッサは、アクセスしているアドレスにベースセクタのアドレスを自動的に追加します。LGDTを使ってこの新しいGDTをインストールした後、この後、3GBで動作するようになりました。これは、プロセッサが参照しているアドレスにcsとdsのセクタのベース（40100000）を追加するからです。例えば、3GBはプロセッサによって1MBに変換され、この例では、 $3GB + \text{ベースセクタ} (40100000) = 1MB$ のフィジカル。

このトリックは非常に簡単に実行でき、うまく機能しますが、64ビット（ロングモード）では機能しません。カーネルはこのトリックを実行した後、ページディレクトリを設定し、自分自身を簡単にマッピングすることができ、その後、ページングを有効にすることができます。

バーチャル・アドレスとマッピング・アドレス

ページングを有効にすると、すべてのメモリ参照が仮想アドレスとして扱われるようになります。これは非常に重要なことです。つまり、ページングを有効にする前に、構造体を適切にセットアップしなければなりません。そうしないと、有効な例外ハンドラがあってもなくても、致命的なトリプルフォールトに陥る可能性があります。

仮想アドレスの形式を覚えていますか？これは、x86の仮想アドレスのフォーマットです。

AAAAAAAA	BBBBBBBB	CCCCCCCCCCCC
ディレクトリインデックス	ページテーブルインデックス	ページへのオフセット

これは非常に重要です。これはプロセッサ（そして私たち）に多くの情報を伝えます。

ディレクトリインデックス部分は、現在のページのディレクトリのどのインデックスを見ればよいかを教えてください。前節のディレクトリエントリ構造のフォーマットを見返してみましょう。各ディレクトリテーブルのエントリには、ページテーブルへのポインタが含まれていることに注目してください。また、このことは同セクションの画像の中でも確認することができます。

ディレクトリテーブル内の各インデックスはページテーブルを指しているため、どのページテーブルにアクセスしているのかわかります。ページテーブルのインデックス部分は、このページテーブル内のどのページエントリにアクセスしているかを教えてください。

各ページエントリは4KBの物理アドレス空間を管理していることを覚えていますか？ページ部分へのオフセットは、このページの物理アドレス空間内のどのバイトを参照しているかを示しています。

ここで起きたことに注目してください。ページテーブルを使って、仮想アドレスを物理アドレスに変換しただけです。そう、とても簡単なのです。何のトリック也没有。

別の例を見てみましょう。仮想アドレス0xc0000000を物理アドレス0x100000にマッピングしたとします。この場合、どうすればいいのでしょうか？先ほどと同じように、0xc0000000が参照している構造体のページを見つける必要があります。ここでは、0xc0000000が仮想アドレスなので、そのフォーマットを見てみましょう。

ズ

ページテーブルのインデックスは、そのページテーブルの中でアクセスしているページであることを覚えていますか？これは**0**なので、最初のページということになります。また、このページのオフセットバイトは**0**であることに注意してください。

アイデンティティ・マッピング

アイデンティティマッピングとは、仮想アドレスを同じ物理アドレスにマッピングすることに他なりません。例えば、仮想アドレス 0x100000 は、物理アドレス 0x100000 にマッピングされます。そう、それだけのことです。実際にこの作業が必要になるのは、最初にページングを設定するときだけです。ページングを有効にしても、現在実行中のコードのメモリアドレスが変わらないようにするためです。これをしないと、すぐにトリプルフォールトになってしまいます。この例は、仮想メモリマネージャの初期化ルーチンで見ることができます。

メモリ管理。インプリメンテーション

インプリメンテーション

私は、まだ見なければならぬ新しいことがいくつかあるので、一度に一つのテーマに集中できるように、ルーチンを小さくするようにしました。

さて、まずはページテーブルとディレクトリテーブルを見てみましょう。

```

//物理アドレスと同様に、私は仮想メモリ用の新しいアドレスタイプ、virtual_addrを作成しました。ページテーブルは、1024
//個のページディレクトリエントリの配列に過ぎません。ことに注意してください。ページディレクトリテーブルも同じですが、これはページディレ
//クトリエントリの配列です。まだ特別なことは何もありません。
//!! i86アーキテクチャでは、テーブルあたり1024エントリが定義されていますが、
//変更されています。
//PAGE_DIRECTORY_INDEX、PAGE_TABLE_INDEX、PAGE_GET_PHYSICAL_ADDRESSは、仮想アドレスのそれぞれの部分を返
//すだけのマクロです。仮想アドレスには特定のページ番号があることを覚えておいてください。これらのマクロによって、仮想アドレスから
//情報を得ることができます。
#define PAGE_DIRECTORY_INDEX(x) (((x) >> 22) & 0x3ff)
#define PAGE_TABLE_INDEX(x) ((x) >> 12) & 0x3ff
#define PAGE_GET_PHYSICAL_ADDRESS(addr) ((addr << 12) & 0xfffff)
//PTABLE_ADDR_SPACE_SIZEは、ページテーブルが表すサイズ（バイト数）を表します。ページテーブルは1024ページで、1ページの
//大きさは4Kなので、1024 * 4K = 4MBとなります。
#define PTABLE_ADDR_SPACE_SIZE (1024 * 0x1000)
//PTABLE_ADDR_SPACE_SIZEは、ページディレクトリが管理するバイト数を表し
//ており、これは仮想アドレス空間のサイズにあたります。1つのページテーブルがアドレス空間の4MBを表し、1つのページディレクトリに
//は1024個のページテーブルが含まれるとすると、4MB * 1024 = 4GBとなります。
//!! ページテーブルは4MBのアドレス空間を表す
#define PTABLE_ADDR_SPACE_SIZE (4 * 1024 * 0x1000)
//ここで紹介する仮想メモリマネージャは、大きなページを扱いません。その代わりに、4Kページのみを管理します。

```

www.brokenthorn.com/Resources/OSDev18.html

私たちが使用している仮想メモリマネージャ（VMM）は、これらの構造に大きく依存しています。ここでは、VMMのルーチンのいくつかを見て、その動作を確認してみましょう。

vmmngr_alloc_page () - 物理メモリにページを確保する

ページを割り当てるために必要なことは、ページが参照する物理メモリの4Kブロックを割り当て、そのブロックからページテーブルのエントリを作成することだけです。

PTE ルーチンを使用すると、この作業が非常に簡単になることに気付きましたか？ 上記は、ページテーブルエントリのPRESENTビットを設定し、そのFRAMEアドレスを、割り当てられたメモリブロックを指すように設定します。このようにして、ページは物理メモリの有効なブロックを指すようになります。物理メモリを割り当てるいいでしょう？

```
void* p = pmmngr_alloc_block ();
```

また、物理アドレスをページに「マッピング」していることにも注目してください。これは、物理アドレスを指すようにページを設定するという事です。したがって、ページはそのアドレスに「マッピング」されます。

vmmngr_free_page () - 物理メモリのページを解放する

```
pt_entry_set_frame (e, (physical_addr)p);
```

```
pt_entry_add_attrib (e, 186, PTE_PRESENT);
```

ページを解放するのはもともと簡単です。当社の物理メモリマネージャを使ってメモリブロックを解放し、ページテーブルのエントリPRESENTビットをクリア（NOT PRESENTと表示）するだけです。

```
return true;
```

これで完成です。vmmngr_free_page () を割り当てる方法と解放する方法ができたので、それらを全ページのテーブルにまとめることができるかどうか見てみましょう...

```
void* p = (void*)pt_entry_pfn
```

vmmngr_ptable_lookup_entry () - ページテーブルからアドレスでページテーブルエントリを取得する

```
pmmngr_free_block (p) です。
```

仮想アドレスからページテーブルエントリ番号を取得する方法ができたので、今度はページテーブルからそれを取得する方法が必要です。このルーチンはまさにそれを行います。このルーチンは、上記の関数を使用して、仮想アドレスをページテーブル配列へのインデックスに変換し、そこからページテーブルエントリを返します。

```
inline pt_entry* vmmngr_ptable_lookup_entry (ptable* p,virtual_addr addr) {...

    if (p)
        return &p->m_entries[ PAGE_TABLE_INDEX (addr)
    ]; return 0;
}
```

このルーチンはポインターを返すので、エントリを必要なだけ変更することができます。いいでしょ？ ページ

テーブルのルーチンはこれでおしまいです。ページングがいかに簡単かわかりますか？)

次は...ページディレクトリのルーチンです。

vmmngr_pdirectory_lookup_entry () - ディレクトリテーブルからアドレスでディレクトリエントリを取得する

仮想アドレスをページディレクトリテーブルのインデックスに変換する方法ができたので、今度はそのインデックスからページディレクトリエントリを取得する方法を提供する必要があります。これは、ページテーブルルーチンの対応と全く同じです。

```
inline pd_entry* vmmngr_pdirectory_lookup_entry (pdirectory* p, virtual_addr addr)

{ if (p)
    return &p->m_entries[ PAGE_TABLE_INDEX (addr) ];
  0を返す。
}
```

vmnmgr_switch_pdirectory () - 新しいページディレクトリに切り替える

これらのルーチンがどれほど小さいかに注目してください。これらのルーチンは、ページテーブルやディレクトリを簡単に操作するための最小限の、しかし非常に効果的なインターフェースを提供します。ページディレクトリをセットアップする際には、それをインストールする方法を提供する必要があります。

前回のチュートリアルでは、ページディレクトリベースレジスター (PDBR) を設定・取得するために、**pmmngr_load_PDBR()**と**pmmngr_get_PDBR()**という2つのルーチンを追加しました。これは、現在のページディレクトリテーブルを格納するレジスタです。x86アーキテクチャでは、PDBRは**cr3**プロセッサ・レジスタです。したがって、これらのルーチンは単に**cr3**レジスタの設定と取得を行います。

vmnmgr_switch_pdirectory ()はこれらのルーチンを使ってPDBRのロードとカレントディレクトリの設定を行います。

```
カレントディレクトリテーブル(グローバル)
pディレクトリ          _cur_directory=0で
※1                    す。
inline bool vmmngr_switch_pdirectory (pdirectory* dir)
{
    if (!dir)
        falseを返す。

    _cur_directory = dir;
    pmmngr_load_PDBR (_cur_pdbr);
    return true;
}

pdirectory* vmmngr_get_directory ()
{
    { return _cur_directory;
}
```

vmnmgr_flush_tlb_entry () - TLBエントリをフラッシュする

TLBが現在のページテーブルをキャッシュしていることを覚えていませんか？時には、TLBや個々のエントリをフラッシュ（無効化）して、現在の値に更新できるようにする必要があります。これはプロセッサによって自動的に行われることもあります（コントロールレジスタを含むmov命令中など）。

プロセッサは、個々のTLBエントリを自分で手動でフラッシュする方法を提供しています。これには、**INVLPG**の指示を受けています。

仮想アドレスを渡すだけで、結果としてページエントリが無効になります。

INVLPGは特権的な命令であるため、これを呼ぶには、**スーパーバイザモード**で動作している必要があります。

vmnmgr_map_page() - マップページ

これは最も重要なルーチンの一つです。このルーチンにより、任意の物理アドレスを仮想アドレスにマッピングすることができます。少し複雑なので、説明します。

パラメータ#0は物理アドレスと仮想アドレスが与えられます。まず最初に行わなければならないのは、この仮想アドレスが置かれているページディレクトリエントリが有効かどうかを確認することです（つまり、以前に割り当てられたことがあり、その**PRESENT**ビットが設定されているかどうか）。

ページディレクトリエントリが設定されているかどうかを確認するには、**PAGE_DIRECTORY_INDEX()**を使ってページディレクトリインデックスを取得し、そのインデックスで、ページディレクトリエントリへのポインタを取得するだけです。そして、**I86_PTE_PRESENT**が設定されているかどうかを確認し、**I86_PTE_PRESENT**が設定されていない場合は、**I86_PTE_PRESENT**をセットして、ページディレクトリ・エントリは存在しないので、それを作成しなければなりません。**I86_PTE_PRESENT** {}。

```

// ! ページテーブルが存在しない場合、それを割り当てる
ptable* table = (ptable*) pmmngr_alloc_block
(); if (!table)
    を返すことができます。

// ! ページテーブルのクリア
memset (table, 0, sizeof(ptable));

// ! 新しいエントリーの作成
pd_entry* entry =
    &pageDirectory->m_entries [PAGE_DIRECTORY_INDEX ( (uint32_t) virt) ]となります。

pd_entry_add_attr (entry, I86_PDE_PRESENT).....これらのビットを有効にする
ために、テーブル内のマップ(*entry | = 3)を行うこともできます。
pd_entry_add_attr (entry, I86_PDE_WRITABLE);
pd_entry_set_frame (entry, (physical_addr)table);
}

```

上記が最初に行うことは、新しいページテーブルのために新しいページを割り当て、それをクリアすることです。その後、再び **PAGE_DIRECTORY_INDEX()** を使用して、仮想アドレスからディレクトリインデックスを取得し、ページディレクトリにインデックスを作成して、ページテーブルエントリへのポインタを取得します。そして、ページテーブルエントリに新しいアロケートページテーブルを指すように設定し、その **PRESENT** ビットと **WRITABLE** ビットを設定して使用できるようにします。

この時点で、ページテーブルはその仮想アドレスで有効であることが保証されます。つまり、ルーチンはそのアドレスをマッピングするだけでよいのです。

```

// ! テーブルの取得
ptable* table = (ptable*) PAGE_GET_PHYSICAL_ADDRESS ( e );

// ! ページ取得
pt_entry* page = &table->m_entries [ PAGE_TABLE_INDEX ( (uint32_t) virt) ] となります。

pt_entry_set_frame ( page, (physical_addr) phys);
pt_entry_add_attr ( page, I86_PTE_PRESENT)を行います。
}

```

上記では、ページテーブルエントリを取得するために、**PAGE_GET_PHYSICAL_ADDRESS()** を呼び出して、ページディレクトリエントリが指し示す物理フレームを取得しています。次に、**PAGE_TABLE_INDEX** を使って仮想アドレスからページテーブルインデックスを取得し、ページテーブルにインデックスを付けてページテーブルエントリを取得します。そして、物理アドレスを指すようにページを設定し、ページの **PRESENT** ビットを設定します。

vmmngr_initialize () - VMMを初期化します。

これは重要なルーチンです。このルーチンは、上記のすべてのルーチンを使用して（というか、ほとんどのルーチンを使用して）、デフォルトのページディレクトリを設定し、インストールし、ページングを有効にします。また、すべてがどのように機能し、どのように組み合わせられるかの例としても使用することができます。このルーチンでは新しいページディレクトリを作成するので、カーネルのために **1MB** の物理データを **3GB** の仮想データにマッピングする必要があります。

これはかなり大きなルーティンなので、分解して見てみましょう。

ページテーブルが **4KB** の物理アドレスに配置されなければならないことを覚えていませんか？物理メモリマネージャ(PMM)のおかげで、**pmmngr_alloc_block()** はすでにこれを実行しているので、心配する必要はありません。割り当てられた **1** つのブロックのサイズはすでに **4K** であるため、ページテーブルのサイズを確保も十分なストレージスペースを持っています（ページテーブルエントリ **1024** 個 * エントリあたり **4** バイト = ページテーブルのサイズ **4K**）。このため、必要なのは **1** つのブロックだけです。

その後、ページテーブルを片手ですべてのページに使用するためにきれいにします。

```

// ! 3GBのページテーブルを確保
// ! 最初の4KBはアドレス0x00000000-pmmngr_alloc_block
for (int i=0; i<frame=0x0, virt=0x00000000; i<1024; i++, frame+=4096, virt+=4096) {...
    を返すことができます。
    // ! 新しいページを作る
// ! ページテーブルをクリアする
vmmngr_ptable_clear (table);

```

```

pt_entry page=0;
pt_entry_add_attrrib (&page, I86_PTE_PRESENT);
pt_entry_set_frame (&page, frame) となります。

// ! ...そして、それをページのテーブルに追加する
table2->m_entries [PAGE_TABLE_INDEX (virt) ] = ページ。
}

```

この部分が少し厄介です。**ページングが有効になると、すべてのアドレスが仮想化されることを覚えていますか？**これが問題になります。この問題を解決するには、仮想アドレスを同じ物理アドレスにマッピングして、同じものを参照するようにしなければなりません。**これが IDENTITY MAPING**です。

上記のコードは、ページテーブルを物理メモリの最初の4MB（ページテーブル全体）にマッピングするものです。新しいページを作成し、そのPRESENTビットにページが参照したいフレームアドレスをセットします。その後、マッピングしている現在の仮想アドレス（frameに格納）をページテーブルインデックスに変換し、そのページテーブルエントリを設定します。

ページテーブルの各ページ（"i"に格納）の"frame"を4K（4096）だけ増やします。（ページテーブルのインデックス0はアドレス0～4093を参照し、インデックス1はアドレス4096を参照する、などと覚えておいてください）。）

ここで問題が発生します。ブートローダーがカーネルを3gbの仮想空間に直接マッピングしてロードするため、カーネルがある領域もリマップする必要があります。

```

1MBから3GBへのマッピング（現在の状況
for (int i=0, frame=0x100000, virt=0xc0000000; i<1024; i++, frame+=4096, virt+=4096)
{...

// ! 新しいページを作る
pt_entry page=0;
pt_entry_add_attrrib (&page, I86_PTE_PRESENT);
pt_entry_set_frame (&page, frame) となります。

// ! ...そして、それをページのテーブルに追加する
table->m_entries [PAGE_TABLE_INDEX (virt) ] = page;
}

```

このコードは、上のループとほとんど同じで、1MBの物理アドレスを3GBの仮想アドレスにマッピングします。これにより、カーネルがアドレス空間にマッピングされ、3GBの仮想アドレスで実行し続けることができるようになります。

```

// ! デフォルトのディレクトリテーブルの作成
pdirectory*dir = (pdirectory*)
pmmngr_alloc_blocks (3); if (!dir)
    を返すことができます。

// ! ディレクトリテーブルをクリアしてカレントに設定する
memset (dir, 0, sizeof (pdirectory));

```

上記は、新しいページディレクトリを作成し、私たちが使用するためにクリアします。

各ページテーブルは、4MBの仮想アドレス空間を表していることを覚えておいてください。各PDEは0x00000000のエントリがページテーブルを指していることを知っています。これは、各ページディレクトリエントリがPDEのPRESENTビットを186_PDE_PRESENTでPDE全体の4GB仮想アドレス空間の中の同じ4MBアドレス空間を表していると書いても差支えないでしょう。entryは186_PDE_WRITEABLEで最初のエントリは最初の4MB、2番目のエントリは次の4MBというように、それぞれに対応しています。今は最初の4MBだけをマッピングしているもので、必要なのは最初のエントリがページテーブルを指すように設定することだけです。

```

pd_entry* entry2 = &dir->m_entries [PAGE_DIRECTORY_INDEX (0x00000000)
] ; pd_entry_add_attrrib (entry2, I86_PDE_PRESENT);
pd_entry_add_attrrib (entry2, I86_PDE_WRITEABLE);
pd_entry_set_frame (entry2, (physical_addr)table2);

```

同様に、3GBのページディレクトリエントリを設定します。これは、カーネルをマッピングするために必要です。

ページディレクトリエントリがPAGEとPRESENTビットも設定していることに注目してください。これにより、ページテーブルが存在し、書き込み可能であることがプロセッサに伝えられます。

```

// ! 現在のPDBRを保存
_cur_pdbr = (physical_addr) &dir->m_entries;

// ! ページのディレクトリに切り替える
vmmngr_switch_pdirectory (dir);

// ! ページングの有効化

```

```
}
    pmmngr_paging_enable (true) です。
}
```

これで、ページディレクトリが設定されたので、ページディレクトリをインストールして、ページングを有効にします。すべてが期待通りに動作していれば、あなたのプログラムはクラッシュしないはずです。動作しない場合は、おそらくトリプルフォールトになるでしょう。

ページの不具合

ご存知のように、ページングを有効にすると、すべてのアドレスが仮想的になります。これらの仮想アドレスはすべて、ページテーブルとページディレクトリのデータ構造に大きく依存しています。これでいいのですが、仮想アドレスがまだ有効でないページにアクセスすることをCPUに要求する場合があります。このような場合、プロセッサから**ページフォルト例外 (#PF)**が発生します。この例外は、ページが**存在しない**とマークされている場合にのみ発生します。**GPF (General Protecton Fault)**は、ページが適切にマッピングされていないにもかかわらず、存在するとマークされ、アクセス可能な場合に発生します。また、ページにアクセスできない場合にも**#GPF**が発生します。

ページフォルトは、CPU割り込み14で、情報を得るためにエラーコードもプッシュされます。プロセッサがプッシュするエラーコードは次のような形式です。

- **ビット0**です。
 - 0: ページが存在していたため、**#PF**が発生した
 - 1: **#PF**が発生したのは、ページが存在していたからではない。
- **ビット1**です
 - 0: **#PF**の原因となった操作が読み取りだった場合
 - 1: **#PF**の原因となった操作が書き込みだった場合
- **ビット2**
 - 0: プロセッサがリング0 (カーネルモード) で動作していた
 - 1: プロセッサがリング3 (ユーザーモード) で動作していた
- **ビット3**
 - 0: 予約済みのビットが書き込まれたため、**#PF**が発生しなかった
 - 1: 予約済みのビットが書き込まれたため、**#PF**が発生した
- **ビット4**
 - 0: **#PF**は命令フェッチ中に発生しなかった
 - 1: **#PF**は命令フェッチ中に発生した

その他のビットはすべて0です。

また、**#PF**が発生した場合、プロセッサはエラーの原因となったアドレスを**CR2**レジスタに格納します。

通常、**#PF**が発生すると、OSは、現在実行中のプログラムの障害アドレスからページをディスクから取得する必要があります。これにはOSのさまざまなコンポーネント (ディスクドライバ、ファイルシステムドライバ、ボリューム/マウントポイントの管理) が必要ですが、私たちはまだ持っていません。このため、ページフォルト処理については、もう少し後に、より進化したOSを手に入れたときに、再び取り上げることにします。

デモ

このデモには、このチュートリアルすべてのソースコードと、それ以上のものが含まれています。このデモには、ブートローダとカーネルの内部にページングコードが含まれており、完全な仮想メモリマネージャ (VMM) を含み、カーネルを自身の仮想アドレス空間内の3GBマークにマッピングします。

今回のデモでは、ビジュアル面での新しさはありません。そのため、新しい写真もありません。しかし、このデモでは、本章で説明したコンセプトを、アセンブリ言語のソース (ブートローダのPaging.asmファイル) とC言語のソース (本章で開発したVMM) の両方で実演しています。

[デモダウンロード](#)

結論

これができて、本当によかったです。このチュートリアルでは、多くの情報をカバーしました。仮想メモリ、仮想アドレスと変換、ページング、方法などです。このチュートリアルでは、まだページングの話をしていません。しかし、ページングについての理解を深め、どのように動作するのか、どのように扱うのかを知ることができたので、今夜は安心して眠りにつくことができます。ほらね。そんなに悪くないでしょう?)

次のチュートリアルでは、キーボードドライバの開発に戻ろうと考えています。すでに出力形式があり、入力も取得できるので、簡単なコマンドラインも作れるかもしれません。)

次の機会まで。

マイク

BrokenThorn Entertainment社。現在、DoEとNeptuneOperating Systemを開発中です。質問やコメントは

ありますか?お気軽に[お問い合わせください](#)。

あなたも記事の改善に貢献したいと思いませんか? もしそうなら、ぜひ[私に教えてください](#)。





オペレーティングシステム開発シリーズ

オペレーティングシステム開発 - キーボード

by Mike, 2009

このシリーズは、オペレーティングシステムの開発を一から実演し、教えることを目的としています。

はじめに

Welcome!

この章では、もう少し複雑なもの、つまりキーボードについて説明します。キーボードの歴史、キーボードの内部構造、8042と8048のマイクロコントローラ、キーボードドライバの開発などについて説明します。

また、このシリーズでプログラムする最初のデバイスになります。ワクワクしますね。ハードウェア・プログラミングの仕組みを学び、経験を積んできましたが、いよいよ実践です。準備はいいですか？これは、1つのコントローラーだけでなく、2つのコントローラーを使ってプログラミングする最初のデバイスでもあります。これらのコントローラーは、お互いに、そして私たちのシステムと通信します。さらに複雑なことに、両方のコントローラーにはそれぞれ独自のコマンドがあり、それらを扱う方法があります。そのため、この章ではいくつかの箇所はかなり詳細に説明しています。

この章には、最初のインタラクティブなデモも含まれています。基本的なコマンドラインパーサーです。興奮しましたか？

また、この章はデバイスドライバーをより深く考察する最初の章でもあります。ハードウェアの抽象化とデバイスドライバーの重要性。

リストをご覧ください。

- キーボード-時代の流れとキーボードレイアウト キ
- ーボードの内部
- キーボードプロトコル キ
- ーボードエンコーダ キー
- ボードコントローラ スキ
- ャンコードセット キーボ
- ードIRQ

頑張りましょう。

キーボード - 歴史

過去に戻る

キーボードは、私たちがコンピューターに入力するための入力装置です。キーボードが登場した当初は、タイプライターをモデルにしていた。しかし、キーボードはタイプライターをそのままモデルにして作られたわけではない。

1877年にクリストファー・レイサム・ショールズがタイプライターの特許を取得すると、いくつかのメーカーや人々がオリジナルのデザインをさらに発展させていった。このような一連の発明を経て完成したのが電信機である。同じ頃、1930年代の中頃には、IBM社が加算機にキーパンチ（タイプライターと組み合わせたパンチカード機）を使っていた。初期のコンピューターのキーボードは、キーパンチと電信機の両方のデザインを採用していた。

ENIAC (Electronic Numerical Integrator And Computer) は、最初の汎用コンピュータである。ENIACは、1946年にパンチカードリーダーを入出力デバイスとして使用した。

1948年、BINAC (BINary Automatic Computer) は、電気機械的に制御されたタイプライターを入力装置と出力装置として使用した。

これらの発明から、キーボードはいつ進化したのでしょうか？私たちが知っているコンピューターのキーボードが現在のように進化したのは、1964年にMIT（フェルナンド・コルバトと共同）、ベル研究所、ゼネラル・エレクトリック社が共同でMultics (Multiplexed Information and Computing Service) というマシンを開発してからです。Multicsでは、新しいインターフェースが登場した。テレビや電動タイプライターに使われているCRT（ブラウン管）の技術を組み合わせて、VDT (Video Display Terminal) を作ったのだ。VDTを使うと、ユーザーは次のようなことができるようになった。