

###### 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、（シャレではなく）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。

次のサイトをお勧めします：

[c programming.com](http://www.cprogramming.com/)

[c plusplus.com](http://www.cplusplus.com/)

[c odecademy.com](https://www.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

T he Great Debate: Which Programming Language Should You Learn First?

*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T itle Pro's Tinkering Guide

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と、どうやって起動すればいいのか分からない。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、何をす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

はじめに

これを読んでくださっているあなたは、すでに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、ポインターはいたるところで使われています。そのため、ポインターを使いこなすことは非常に重要です。

ポインターとは、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。ポインターを定義するには、\* 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。ワイルドポインターと は、何でも指し示すことができるポインターのことです。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、この同じコードを将来の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

ポインターは、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み出そうとすると、ゴミ（その時点でシステムデータバス上にあったもの）が表示 されます。存在しないメモリアドレスに書き込もうとしても何も起こりません。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、ポインターを使って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に書き込むのと同じ 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、ポインターは、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。ポインターは、「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」などと言えばいいのです。

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 ; disable interrupts

また、アセンブリコードのブロックもあります。

\_asm {

cli hlt

}

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ま す。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も使用できます。

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。現在開発中の「DoE」と「N* eptune オペレーティングシステム

*ご質問やご意見がございましたら お気軽にお問い合わせください。*

記事をより良くするために、あなたも貢献してみませんか？よろしければ、ぜひご連絡ください。



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 - 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」をご覧になってください。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可能 性があります。

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、以下を参考にしてみて ください。

L earn C - Cプログラミング初心者のための完全なリソースです。

C++で考えてみる

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。

[A ssembly Language: Step by Step](http://www.duntemann.com/assembly.htm) [A rt of Assembly](http://www.arl.wustl.edu/~lockwood/class/cs306/books/artofasm/toc.html)

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はさよならを言いましょう。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もここから入手することができます。

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

このリストに追加したい方は、私までご連絡ください。

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ださい。

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。

このページ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 [address] [drive] [firstsector] [number]

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。

C:\Documents and Settings\Michael>debug boot\_loader.bin

-w 100 0 0 1

-q

このコマンドの詳細については、このチュートリアルを参照してください。

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？

このページ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。

**Thats all, fokes**

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起動可能な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。

1. 全ての設定を行う
2. 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。Bochs Emulatorを設定し、フロッピーイメージから起動する。

1. ブートローダ

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。

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。

1. カーネル（そして基本的には他のすべてのプログラム）
2. 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。

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N eptune Operating Systemを開発中。*

*ご質問やご意見がございましたら お気軽にお問い合わせください。*

記事をより良くするために、あなたも貢献してみませんか？よろしければ、ぜひご連絡ください。



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

素晴らしく、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・ システムの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るとしたら（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、移植性や互換性もありません。

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するものです。

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月に発売することを余儀なくされた。

そのバージョンは

BOS/360 - 8KB Configuation.

DOS/360 - 16KB Configuation with disk. TOS/360 - 16KB Configuation with tape.

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。

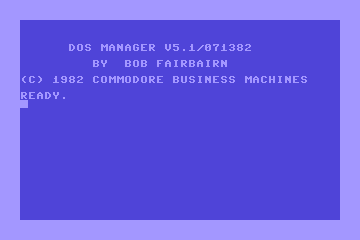
1969年、Unixだ!

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。

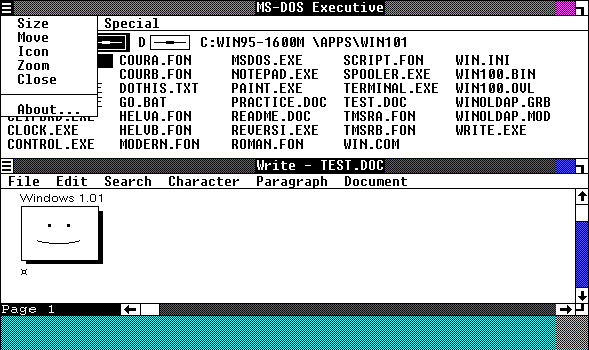
Unixには、カーネル、ファイルシステム、コマンドシェルがあります。コマンドシェルを利用してOSと対話するGUI（グラフィカル・ユー ザー・インターフェース）も数多く存在し、より親しみやすく美しい外観を実現しています。

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。

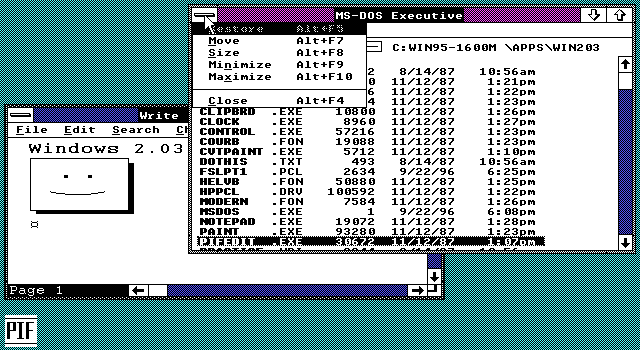
1985年 - Microsoft Windows 1.0



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る。しかし、どの

「ウィンドウ」も重なることができなかったので、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



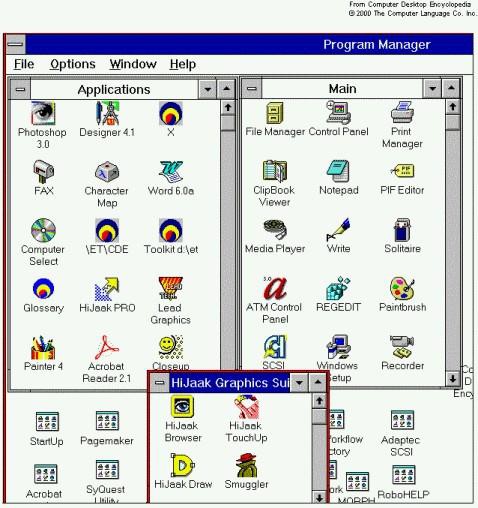
Windowsの第2版は、DOSのグラフィカルシェルでありながら、ウィンドウの重なりや色数の増加をサポートしていた。しかし、DOSの限界の ため、あまり使われなかった。

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りませんでした。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かったが、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。

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タラクションのことです。

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, [09000h]

; There is no such thing as Access Violations in OS Development

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

|  |  |  |  |
| --- | --- | --- | --- |
| mov mov | bx, [07bffh] cx, 10 | ; | or some other address less then 7c00h |
| .loop1: |  |  |  |
| mov | [bx], 0h | ; | clear bx |
| inc | bx | ; | go to next address |
| loop | .loop1 | ; | loop until cx=0 |

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、より簡潔に定義できるようになるはずです。

*"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。



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、コンピューターの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、コンピューターの基礎となる「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Interupt 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で使用される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。

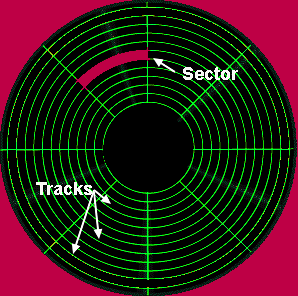
セクター

セクター」とは、簡単に言えば、512バイトのグロープを表します。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ある。

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。

|  |  |  |
| --- | --- | --- |
| mov | ax, 4ch |  |
| inc | bx | ; 512 byte |
| mov | [var], bx | ; 514 byte |

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。

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）。

; enable interrupts--we're in the clear!

sti

; do something...

; clear interrupts

cli

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せんか？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

Yiippee! \*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  ; - A Simple Bootloader  ;  ; Operating Systems Development Tutorial  ;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* | | | |
| org | | 0x7c00 | ; We are loaded by BIOS at 0x7C00 |
| bits 16 | |  | ; We are still in 16 bit Real Mode |
| Start: | |  |  |
| cli hlt | |  | ; Clear all Interrupts  ; halt the system |

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。

org

0x7c00

; We are loaded by BIOS at 0x7C00

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ます。つまり、最初の命令は0x7C00になるということです。

bits 16

; We are still in 16 bit Real Mode

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。

コンピューターを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。

times 510 - ($-$$) db 0

; We have to be 512 bytes. Clear the rest of the bytes with 0

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

; Boot Signiture

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はブートローダ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。

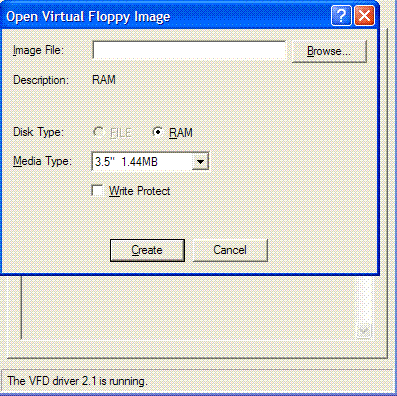
アセンブル後、"Boot1.bin "という正確な512バイトのファイルができるはずです。

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。

VFD（バーチャル・フロッピー・ドライブ）の使い方

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

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、ブートローダ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。

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 file first\_byte last\_byte drive

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 などを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 and VGA BIOS images romimage: file=BIOS-bochs-latest, address=0xf0000  vgaromimage: VGABIOS-lgpl-latest  # boot from floppy using our disk image -------------------------------  floppya: 1\_44=a:, status=inserted # Boot from drive A  # logging and reporting | | |
| log: error: info: | OSDev.log action=report action=report | # All errors and info logs will output to OSDev.log |

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の位置を変更するには、この行を変更する必要があります。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 0xFFFFF - 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。

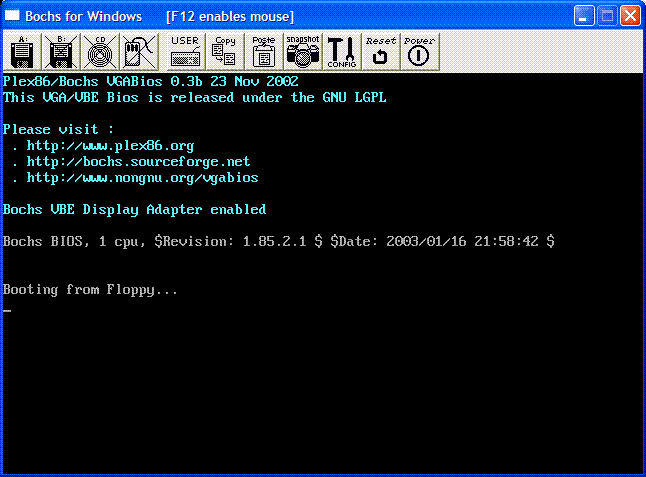
このステップを行う必要がある場合とない場合があります。もし、Biosが0xFFFFFで終わらなければならないというエラーがBochsから出た場合は、このステップを完了する必要があり、動作するはずです。

Bochsの使い方

1. Bochsを使うために。
2. bochs.exeの実行
3. オプション2（Read options form）を選択し、エンターキーを押します。
4. 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。

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、ボー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N eptune Operating Systemを開発中 質問やコメントは？お気軽にご連絡ください。*

記事をより良くするために、あなたも貢献してみませんか？よろしければ、ぜひご連絡ください。



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こるのか、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なものにするために、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？*

##### プロセッサーモード

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プロセッサーは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 :) ところで、リアルモードのどこがリアルなのでしょうか？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

今回は、チュートリアル2をご紹介します。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」プロセッサー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というプロセッサーを開発・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ー8086は、8088が発売されるまで、すでに16ビットプロセッサー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。

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 トを使用しています。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？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。

それがすべてです :)

Absolute (Exact) Memory Address = (Segment Address \* 16(decimal)) + Offset

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。上記を計算式で表すと、絶対アドレス0x7C00に変換できます。

07C0:0000 < 07C0 is the segment, and 0 is the offset

base address = base address \* segment size (16) + offset 07C0:0000 = 07C0 \* 16 (decimal) + 0

= 07C00 + 0 = 0x7C00

セグメント：オフセット問題

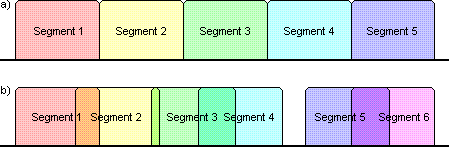
セグメント：オフセットは非常にユニークです。セグメントとオフセットの値を変えることで、異なるセグメント：オフセットのペアが同じ絶対アドレスを出力することがわかります。なぜでしょう？それは、どちらも同じメモリロケーションを参照しているからです。

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。記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。

##### プロセッサーモードの切り替え

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「Potected 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などの記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」「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ね？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を | | | | | |
| 示します。 | | | | | |
| ;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* | | | | | |
| ; Boot1.asm | | | | | |
| ; - A Simple Bootloader | | | | | |
| ; | | | | | |
| ; Operating Systems Development Tutorial | | | | | |
| ;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* | | | | | |
| bits | 16 | ; | We are still | in 16 bit | Real Mode |
| org | 0x7c00 | ; | We are loaded | by BIOS | at 0x7C00 |
| start: | jmp loader | ; | jump over OEM | block |  |

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;

; OEM Parameter block

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;

|  |  |
| --- | --- |
| TIMES 0Bh-$+start DB 0  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 |
| bsUnused: | DB 0 |
| bsExtBootSignature: | DB 0x29 |

bsSerialNumber: bsVolumeLabel: bsFileSystem:

DD 0xa0a1a2a3 DB "MOS FLOPPY " DB "FAT12 "

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;

; Bootloader Entry Point

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;

loader:

cli hlt

; Clear all Interrupts

; halt the system

times 510 - ($-$$) db 0

; We have to be 512 bytes. Clear the rest of the bytes with 0

dw 0xAA55

; Boot Signiture

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  mov | bx, bx  ah, 0x0e | ; A faster method of clearing BX to 0 |
| mov | al, 'A' |  |
| 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

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Prints a string

; DS=>SI: 0 terminated string

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Print:

PrintDone:

ret

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;

; Bootloader Entry Point

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*; loader:

; Error Fix 1

times 510 - ($-$$) db 0

; We have to be 512 bytes. Clear the rest of the bytes with 0

dw 0xAA55

; Boot Signiture

|  |  |  |
| --- | --- | --- |
|  |  |  |
| or | al, al | ; al=current character |
| jz | PrintDone | ; null terminator found |
| mov | ah, 0eh | ; get next character |
| int | 10h |  |
| jmp | Print |  |

|  |  |  |
| --- | --- | --- |
| xor | ax, ax | ; Setup segments to insure they are 0. Remember that |
| mov | ds, ax | ; we have ORG 0x7c00. This means all addresses are based |
| mov | es, ax | ; from 0x7c00:0. Because the data segments are within the same  ; code segment, null em. |
| mov call | si, msg Print |  |
| cli hlt |  | ; Clear all Interrupts  ; halt the system |

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リ。以下に例を示します。

; Now AX = Amount of KB in system recorded by BIOS

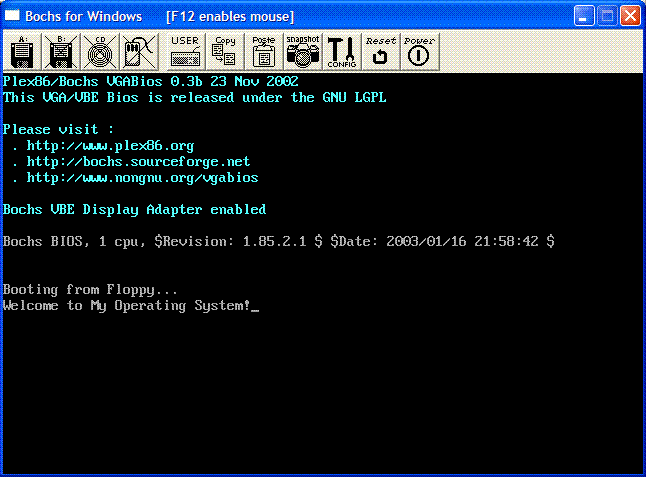
ax, ax 0x12

xor int

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。

**Demo**



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  ; Boot1.asm  ; - A Simple Bootloader  ;  ; Operating Systems Development Tutorial  ;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* | | | | | |
| bits | 16 | ; | We are still | in 16 bit | Real Mode |
| org | 0x7c00 | ; | We are loaded | by BIOS | at 0x7C00 |
| start: | jmp loader | ; | jump over OEM | block |  |

; \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;

; OEM Parameter block

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;

; Error Fix 2 - Removing the ugly TIMES directive -------------------------------------

;; TIMES 0Bh-$+start DB 0 ; The OEM Parameter Block is exactally 3 bytes

; from where we are loaded at. This fills in those

; 3 bytes, along with 8 more. Why?

bpbOEM db "My OS " ; This member must be exactally 8 bytes. It is just

; the name of your OS :) Everything else remains the same.

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

bsUnused: DB 0

bsExtBootSignature: DB 0x29 bsSerialNumber: DD 0xa0a1a2a3

bsVolumeLabel: DB "MOS FLOPPY "

bsFileSystem: DB "FAT12 "

msg db "Welcome to My Operating System!", 0 ; the string to print

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Prints a string

; DS=>SI: 0 terminated string

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

|  |  |  |
| --- | --- | --- |
| Print: |  |  |
| lodsb |  | ; load next byte from string from SI to AL |
| or | al, al | ; Does AL=0? |
| jz | PrintDone | ; Yep, null terminator found-bail out |
| mov | ah, 0eh | ; Nope-Print the character |
| int | 10h |  |
| jmp | Print | ; Repeat until null terminator found |
| PrintDone: |  |  |
| ret |  | ; we are done, so return |

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;

; Bootloader Entry Point

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*; loader:

times 510 - ($-$$) db 0

; We have to be 512 bytes. Clear the rest of the bytes with 0

dw 0xAA55

; Boot Signiture

|  |  |  |
| --- | --- | --- |
| xor | ax, ax | ; Setup segments to insure they are 0. Remember that |
| mov | ds, ax | ; we have ORG 0x7c00. This means all addresses are based |
| mov | es, ax | ; from 0x7c00:0. Because the data segments are within the same  ; code segment, null em. |
| mov | si, msg | ; our message to print |
| call | Print | ; call our print function |
| xor | ax, ax | ; clear ax |
| int | 0x12 | ; get the amount of KB from the BIOS |
| cli hlt |  | ; Clear all Interrupts  ; halt the system |

##### 結論

よくぞここまで来れ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の4種類があります。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読を行います。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 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、プロセッサー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、ファーコール、ファーリターンなど、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することができます。

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 :) さあ、あなたの希望は

:)

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

ディスクのアクセスには、BIOSの割り込み0x13を使用します。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も、すぐにディスクの最初のセクター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。

|  |  |  |
| --- | --- | --- |
| .Reset:  mov | ah, 0 | ; reset floppy disk function |
| mov | dl, 0 | ; drive 0 is floppy drive |
| int | 0x13 | ; call BIOS |
| jc | .Reset | ; If Carry Flag (CF) is set, there was an error. Try resetting again |

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

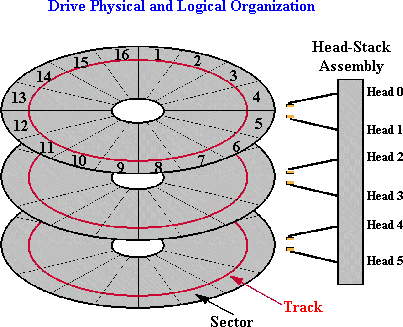
DL = ドライブ番号（ハードディスクの場合はビット7を設定） ES:BX = セクタを読み出すバッファ戻ります。

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。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。

CH = Low eight bits of cylinder number



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 = Sector Number (Bits 0-5). Bits 6-7 are for hard disks only

これが最初のセクターで、そこから読み始めます。覚えておいてください。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の間にしかなりません。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 = Head Number

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いてください。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。

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 = Drive Number (Bit 7 set for hard disks) ES:BX = Buffer to read sectors to

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。

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| .Reset: |  | |  | | | | | | |
| mov  mov | ah,  dl, | 0  0 | ; reset floppy disk function  ; drive 0 is floppy drive | | | | | | |
| int jc | 0x13  .Reset | | ; call BIOS  ; If Carry Flag (CF) is set, there was an error. Try resetting again | | | | | | |
| mov mov xor | ax,  es,  bx, | 0x1000  ax bx | ; we are going to read sector to into address 0x1000:0 | | | | | | |
| .Read: |  | |  | | | | | | |
| mov | ah, | 0x02 | ; | function 2 |  |  |  |  |  |
| mov | al, 1 | | ; | read 1 sector |  |  |  |  |  |
| mov | ch, 1 | | ; | we are reading the second sector past us, | so its | still | on | track | 1 |
| mov | cl, 2 | | ; | sector to read (The second sector) |  |  |  |  |  |
| mov | dh, 0 | | ; | head number |  |  |  |  |  |
| mov | dl, 0 | | ; | drive number. Remember Drive 0 is floppy | drive. |  |  |  |  |
| int | 0x13 | | ; | call BIOS - Read the sector |  |  |  |  |  |
| jc | .Read | | ; | Error, so try again |  |  |  |  |  |
| jmp | 0x1000:0x0 | | ; | jump to execute the sector! |  |  |  |  |  |

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と、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: bpbSectorsPerCluster: bpbReservedSectors: | DW 512  DB 1  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 |
| bsUnused: | DB 0 |
| bsExtBootSignature: | DB 0x29 |
| bsSerialNumber: | DD 0xa0a1a2a3 |
| bsVolumeLabel: | DB "MOS FLOPPY " |
| bsFileSystem: | 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: bpbNumberOfFATs:

DW 1

DB 2

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。 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: bpbSectorsPerFAT:

DB 0xF0 DW 9

bpbRootEntries: bpbTotalSectors:

DW 224

DW 2880

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。bpbHeadsPerCylinderは、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。

bpbHiddenSectors:

DD 0

bpbSectorsPerTrack: bpbHeadsPerCylinder:

DW 18

DW 2

bpbTotalSectorsBig: bsDriveNumber:

DD 0

DB 0

bsUnused: bsExtBootSignature:

DB 0 DB 0x29

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。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？

Boot Signiture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: bsVolumeLabel: bsFileSystem:

DD 0xa0a1a2a3

DB "MOS FLOPPY " DB "FAT12 "

シリアルナンバーはフォーマットした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。

Low 16 bits = ((seconds + month) << 8) + (hundredths + day\_of\_month) High 16 bits = (hours << 8) + minutes + year

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 Lable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ず、それ以上でもそれ以下でもありません。

**Demo**

わぁ、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。

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

;

;

;

;

Boot1.asm

- A Simple Bootloader

Operating Systems Development Tutorial

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

bits

16

; We are still in 16 bit Real Mode

|  |  |  |
| --- | --- | --- |
| org | 0x7c00 | ; We are loaded by BIOS at 0x7C00 |
| start: | jmp loader | ; jump over OEM block |

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;

; OEM Parameter block / BIOS Parameter Block

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;

TIMES 0Bh-$+start DB 0 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

bsUnused: DB 0

bsExtBootSignature: DB 0x29 bsSerialNumber: DD 0xa0a1a2a3

bsVolumeLabel: DB "MOS FLOPPY "

bsFileSystem: DB "FAT12 "

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Prints a string

; DS=>SI: 0 terminated string

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

|  |  |  |  |
| --- | --- | --- | --- |
| Print: | lodsb |  | ; load next byte from string from SI to AL |
| PrintDone: | or jz mov int jmp  ret | al, al PrintDone ah, 0eh 10h  Print | ; Does AL=0?  ; Yep, null terminator found-bail out  ; Nope-Print the character  ; Repeat until null terminator found  ; we are done, so return |

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;

; Bootloader Entry Point

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;

|  |  |  |  |
| --- | --- | --- | --- |
| loader: |  | | |
| .Reset: | mov | ah, 0 | ; reset floppy disk function |
|  | mov int jc | dl, 0 0x13  .Reset | ; drive 0 is floppy drive  ; call BIOS  ; If Carry Flag (CF) is set, there was an error. Try resetting again |
|  | mov mov xor | ax, 0x1000 es, ax bx, bx | ; we are going to read sector to into address 0x1000:0 |
|  | mov mov mov mov mov mov int | ah, 0x02 al, 1  ch, 1  cl, 2  dh, 0  dl, 0 0x13 | ; read floppy sector function  ; read 1 sector  ; we are reading the second sector past us, so its still on track 1  ; sector to read (The second sector)  ; head number  ; drive number. Remember Drive 0 is floppy drive.  ; call BIOS - Read the sector |
|  | jmp | 0x1000:0x0 | ; jump to execute the sector! |

times 510 - ($-$$) db 0 ; We have to be 512 bytes. Clear the rest of the bytes with 0

dw 0xAA55 ; Boot Signiture

; End of sector 1, beginning of sector 2 ---------------------------------

org 0x1000 ; This sector is loaded at 0x1000:0 by the bootsector

cli ; just halt the system

hlt

##### 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。

~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 - 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」という概念を作るためのものです。

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 contigous）、そのファイルはフラグメンテーションされているといいます。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floppr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、Thr 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 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は非常によく似ています。

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

FAT12とその仕組みを理解するには、典型的なフォーマット済みディスクの構造を見るのがよい。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Boot Sector | Extra Reserved Sectors | File Allocation Table 1 | File Allocation Table 2 | Root Directory (FAT12/FAT16 Only) | Data Region containng files and directories. |

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ローダ)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も注意してください。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ローダが含まれています。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。

**Value marks free cluster :** 0x00

**Value marks Reserved cluster :** 0x01

**This cluster is in use--the value represents next cluster :** 0x002 through 0xFEF

**Reserved values :** 0xFF0 through 0xFF6

**Value marks bad clusters :** 0xFF7

**Value marks this cluster as the last in the file :** 0xFF8 through 0xFFF

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0xfef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。そのセクタ数をBIOS Paramete Blockで定義します。

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の開始クラスタ 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。この番号を上のリストと比較して、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。

**Bytes 0-7 : DOS File name (Padded with spaces) Bytes 8-10 : DOS File extension (Padded with spaces) Bytes 11 :** File attributes. This is a bit pattern:

**Bit 0 :** Read Only **Bit 1 :** Hidden **Bit 2 :** System

**Bit 3 :** Volume Label

**Bit 4 :** This is a subdirectory

**Bit 5 :** Archive

**Bit 6 :** Device (Internal use)

**Bit 6 :** Unused

**Bytes 12 :** Unused

**Bytes 13 :** Create time in ms

**Bytes 14-15 :** Created time, using the following format:

**Bit 0-4 :** Seconds (0-29)

**Bit 5-10 :** Minutes (0-59)

**Bit 11-15 :** Hours (0-23)

**Bytes 16-17 :** Created year in the following format:

**Bit 0-4 :** Year (0=1980; 127=2107

**Bit 5-8 :** Month (1=January; 12=December)

**Bit 9-15 :** Hours (0-23)

**Bytes 18-19 :** Last access date (Uses same format as above) **Bytes 20-21 :** EA Index (Used in OS/2 and NT, dont worry about it) **Bytes 22-23 :** Last Modified time (See byte 14-15 for format) **Bytes 24-25 :** Last modified date (See bytes 16-17 for format) **Bytes 26-27 : First Cluster**

**Bytes 28-32 : File Size**

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。

**Bytes 0-7 : DOS File name (Padded with spaces) Bytes 8-10 : DOS File extension (Padded with spaces)**

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" (Note the padding!)

### 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: | 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 |
| bsUnused: | DB | 0 |
| bsExtBootSignature: | DB | 0x29 |
| bsSerialNumber: | DD | 0xa0a1a2a3 |
| bsVolumeLabel: | DB | "MOS FLOPPY " |
| bsFileSystem: | DB | "FAT12 " |

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に見たことがあるものばかりです。

;

DS=>SI: 0 terminated string

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;

Print:

lodsb or

jz mov int jmp

; load next byte from string from SI to AL al, al ; Does AL=0?

PrintDone

ah, 10h Print

0eh

; Yep, null terminator found-bail out

; Nope-Print the character

; Repeat until null terminator found

PrintDone:

ret

; we are done, so return

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;

; Second Stage Loader Entry Point

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;

; Note: Here, we are executed like a normal

; COM program, but we are still in Ring 0.

; We will use this loader to set up 32 bit

; mode and basic exception handling

; This loaded program will be our 32 bit Kernel.

; We do not have the limitation of 512 bytes here,

; so we can add anything we want here!

org 0x0

; offset to 0, we will set segments later

bits 16

; we are still in real mode

; we are loaded at linear address 0x10000

jmp main

; jump to main

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;

; Prints a string

|  |  |  |
| --- | --- | --- |
| main: |  |  |
| cli |  | ; clear interrupts |
| push pop | cs ds | ; Insure DS=CS |
| mov call | si, Msg Print | |
| cli hlt |  | ; clear interrupts to prevent triple faults  ; hault the system |
| ;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;  ; Data Section  ;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;  Msg db "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。

WORD [bpbRootEntries] ; number of root entrys

WORD [bpbBytesPerSector] ; get sectors used by root directory

; 32 byte directory entry

ax, 0x0020

mov mul div

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いてください。 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。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Boot Sector | Extra Reserved Sectors | File Allocation Table 1 | File Allocation Table 2 | Root Directory (FAT12/FAT16 Only) | Data Region containng files and directories. |

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・・・。

; Now, AX = starting sector of root directory

al, [bpbNumberOfFATs] ; Get number of FATs (Useually 2)

[bpbSectorsPerFAT] ; number of FATs \* sectors per FAT; get number of sectors ax, [bpbReservedSectors] ; add reserved sectors

mov mul add

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。

mov bx, 0x0200 ; load root directory to 7c00:0x0200 call ReadSectors

ルートディレクトリ - 完全な例

このサンプルコードは、チュートリアルの最後に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:

; compute size of root directory and store in "cx"

; compute location of root directory and store in "ax"

mov mul add mov add

al, BYTE [bpbNumberOfFATs] ; number of FATs

WORD [bpbSectorsPerFAT] ; sectors used by FATs ax, WORD [bpbReservedSectors] ; adjust for bootsector WORD [datasector], ax ; base of root directory WORD [datasector], cx

; read root directory into memory (7C00:0200)

mov call

bx, 0x0200 ReadSectors

; copy root dir above bootcode

|  |  |  |
| --- | --- | --- |
| xor | cx, cx |  |
| xor | dx, dx |  |
| mov | ax, 0x0020 | ; 32 byte directory entry |
| mul | WORD [bpbRootEntries] | ; total size of directory |
| div | WORD [bpbBytesPerSector] | ; sectors used by directory |
| xchg | ax, cx |  |

###### 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...

; browse root directory for binary image

mov mov

.LOOP:

cx, [bpbRootEntries]

; the number of entrys. If we reach 0, file doesnt exist

di, 0x0200

; Root directory was loaded here

|  |  |  |
| --- | --- | --- |
| push | cx |  |
| mov | cx, 11 | ; eleven character name |
| mov | si, ImageName | ; compare the 11 bytes with the name of our file |
| push | di |  |
| rep cmpsb |  | ; test for entry match |
| pop | di |  |
| je | LOAD\_FAT | ; they match, so begin loading FAT |
| pop | cx |  |
| add | di, 32 | ; they dont match, so go to next entry (32 bytes) |
| loop | .LOOP |  |
| jmp | FAILURE | ; no more entrys left, file doesnt exist :( |

次のステップへ...。

###### ステップ3: FATの読み込み

ステップ1：スタートクラスターの取得

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。

; Yippe--dx now stores the starting cluster number

dx, [di + 0x001A] ; di contains starting address of entry. Just refrence byte 26 (0x1A) of entry

mov

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...

DB 2

DW 9

bpbNumberOfFATs: bpbSectorsPerFAT:

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 |  |
| mov | al, [bpbNumberOfFATs] | ; number of FATs |
| mul | WORD [bpbSectorsPerFAT] | ; multiply by number of sectors per FAT |
| ; ax = number of sectors the FATs use! | |  |

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で あとは...読んでみてください :)

|  |  |  |
| --- | --- | --- |
| mov | bx, 0x0200 | ; address to load to |
| call | ReadSectors | ; load the FAT table |

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:

; save starting cluster of boot image

; compute size of FAT and store in "cx"

; compute location of FAT and store in "ax"

mov

ax, WORD [bpbReservedSectors]

; adjust for bootsector

; read FAT into memory (7C00:0200)

mov call

bx, 0x0200 ReadSectors

; copy FAT above bootcode

|  |  |  |
| --- | --- | --- |
| mov  call | si, msgCRLF  Print |  |
| mov | dx, WORD [di + 0x001A] |  |
| mov | WORD [cluster], dx | ; file's first cluster |

|  |  |  |
| --- | --- | --- |
| xor | ax, ax |  |
| mov | al, BYTE [bpbNumberOfFATs] | ; number of FATs |
| mul | WORD [bpbSectorsPerFAT] | ; sectors used by FATs |
| mov | cx, ax |  |

###### 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

=

(cluster - 2 ) \* sectors per cluster

それはとても簡単なことです。） 例を挙げてみましょう。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| sub | | ax, 0x0002 | ; subtract 2 from cluster number | |
| xor | | cx, cx |  | |
| mov | | cl, BYTE [bpbSectorsPerCluster] | | | ; get sectors per cluster | |
| mul | | cx | | | ; multply | |

LBAからCHSへの変換

これは少し複雑ですが、それでも非常に簡単です。

|  |  |  |
| --- | --- | --- |
| absolute sector | = | (LBA % sectors per track) + 1 |
| absolute head | = | (LBA / sectors per track) % number of heads |
| absolute track | = | LBA / (sectors per track \* number of heads) |

その一例をご紹介しましょう。

; not much else to do :)

BYTE [absoluteTrack], al

mov ret

; prepare dx:ax for operation

; mod by number of heads (Absolue head formula)

; everything else was already done from the first formula

dx, dx

WORD [bpbHeadsPerCylinder] BYTE [absoluteHead], dl

xor div mov

; these forumlas are very simular...

; prepare dx:ax for operation

; divide by sectors per track

; add 1 (obsolute sector formula)

dx, dx

WORD [bpbSectorsPerTrack] dl

BYTE [absoluteSector], dl

xor div inc mov

LBACHS:

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。

; sectors to read

; read in cluster

; cluster to read

; buffer to read into

; convert cluster to LBA

ax, [cluster] bx

ClusterLBA cx, cx

cl, [bpbSectorsPerCluster]

ReadSectors bx

mov pop call xor mov call push

###### 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ということは、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。

|-0 cluster ----| |2nd cluster---| |4th cluster |

|

|

|1st cluster | |3rd cluster-|

|

|

|

01110101 00111101 0011101 0111010 0011110 0011110

0111010

|

01011101

|

|

|

Note: Binary numbers seperated in bytes. Each 12 bit FAT cluster entry is displayed.

|

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ーし ますが、第2バイトのすべてをオコピー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）。

; compute next cluster

mov

ax, WORD [cluster] ; identify current cluster from FAT

; is the cluster odd or even? Just divide it by 2 and test!

|  |  |  |
| --- | --- | --- |
| mov | cx, ax | ; copy current cluster |
| mov | dx, ax | ; copy current cluster |
| shr | dx, 0x0001 | ; divide by two |
| add | cx, dx | ; sum for (3/2) |
| mov | bx, 0x0200 | ; location of FAT in memory |
| add | bx, cx | ; index into FAT |
| mov | dx, WORD [bx] | ; read two bytes from FAT |

test ax, 0x0001

jnz .ODD\_CLUSTER

; Remember that each entry in the FAT is a 12 but value. If it represents

; a cluster (0x002 through 0xFEF) then we only want to get those 12 bits

; that represent the next cluster

.EVEN\_CLUSTER:

and jmp

dx, 0000111111111111b

.DONE

; take low twelve bits

.ODD\_CLUSTER:

shr dx, 0x0004

; take high twelve bits

.DONE:

mov cmp jb

WORD [cluster], dx dx, 0x0FF0

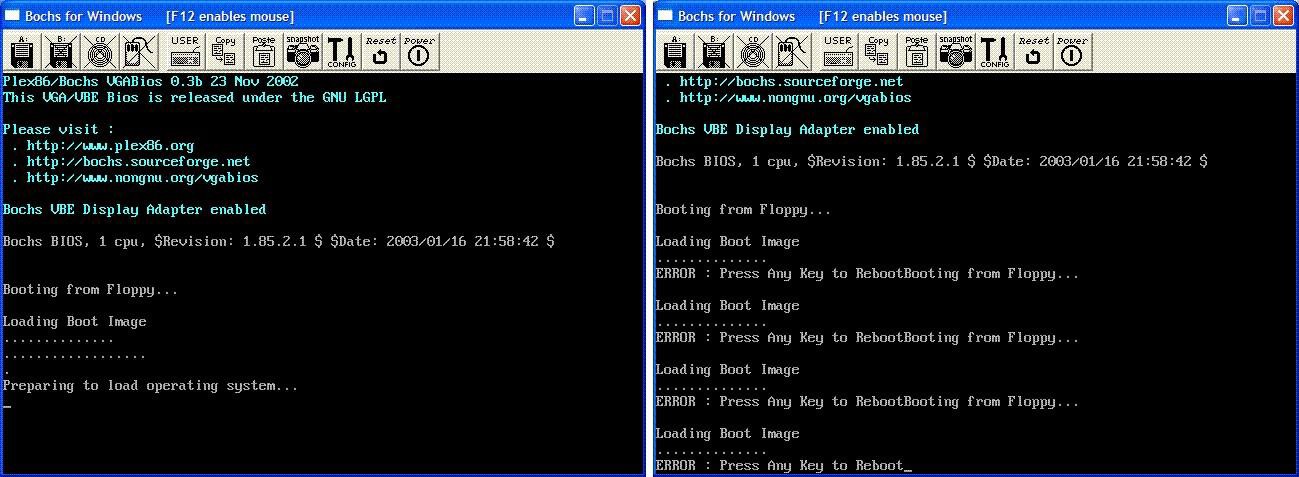
; store new cluster

; test for end of file

LOAD\_IMAGE

; we are not done yet--go to next cluster

**Demo**



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N eptune Operating Systemを開発中 質問やコメントは？ お気軽にご連絡ください。*

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アーキテクチャを詳細に見ていきます。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が、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。

The 80x86 Registers System Orginazation The System Bus

Real Mode Memory Map How an instruction executes 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、アーキテクチャー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。これは、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るのです。

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。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。

Has **Memory Protection**

Has hardware support for **Virtual Memory** and **Task State Switching (TSS)**

Hardware support for interruting programs and executing another

4 Operating Modes: **Ring 0, Ring 1, Ring 2, Ring 3**

Access to 32 bit registers Access to up to 4 GB of memory

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（Useually）であるのに対し、我々はRing 0であることを覚えて

おいてください。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。

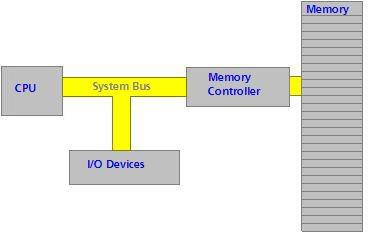
システムアーキテクチャやプロセッサの動作を理解することで、この問題をより深く理解することができます。

### 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） 例えば。



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。しかし、ここではほとんどのアセンブリー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ビットのプロセッサは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、64ビットのデータタイプで、32ビットのデータバスがあるとします。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ビットプロセッサー」「16ビットプロセッサー」などの用語は、データバスのサイズを指します。つまり、「32ビットプロセッサー」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。

メモリーアドレス」とは、メモリーコントローラー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、それをメモリロケーションとして解釈します。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ケートします。

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つの値しかないため、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いてください。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に制限されていました。

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です。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コントローラーに接続されており、I/Oコントローラーがデバ

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る役割を担っています。

"コントローラー"・・・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ーは、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。コントローラー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ーとは何でしょうか？

コントローラ

コントローラーは、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。(コントローラー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ーは、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コントローラーは、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。

デュアルチャンネルコントローラー

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よいのでしょうか。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ー

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ー

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ー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ー」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ー／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マルチプレクサ回路は？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、ジーヴスのためにも、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、待ってください 。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ーを表しています 。そんな感じです。

You may know that several port numbers could represent the same controller. The reason? **Memory Mapped I/O**. The basic idea is that we communicate to hardware by specifying certain memory addresses. **The port number represents this address** Once more, kind of. The

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。詳しくは後ほどご紹介します。

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** - Real Mode Interrupt Vector Table

**0x00000400 - 0x000004FF** - BIOS Data Area

**0x00000500 - 0x00007BFF** - Unused

**0x00007C00 - 0x00007DFF** - Our Bootloader

**0x00007E00 - 0x0009FFFF** - Unused

**0x000A0000 - 0x000BFFFF** - Video RAM (VRAM) Memory

**0x000B0000 - 0x000B7777** - Monochrome Video Memory

**0x000B8000 - 0x000BFFFF** - Color Video Memory

**0x000C0000 - 0x000C7FFF** - Video ROM BIOS

**0x000C8000 - 0x000EFFFF** - BIOS Shadow Area

**0x000F0000 - 0x000FFFFF** - System 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で、特定のアドレスに読み書きすることで、コンピューター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、各コントローラーが聞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は割り込みベクターテーブルを設定し、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。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Default x86 Port Address Assignments** | | | | |
| Address Range | First QWORD | Second QWORD | Third QWORD | Fourth QWORD |
| 0x000-  0x00F | DMA Controller Channels 0-3 | | | |
| 0x010-  0x01F | System Use | | | |
| 0x020-  0x02F | Interrupt Controller 1 | System Use | | |
| 0x030-  0x03F | System Use | | | |
| 0x040-  0x04F | System Timers | System Use | | |
| 0x050-  0x05F | System Use | | | |
| 0x060-  0x06F | Keyboard/PS2 Moude (Port 0x60)  Speaker (0x61) | Keyboard/PS2 Mouse (0x64) | System Use | |
| 0x070-  0x07F | RTC/CMOS/NMI (0x70, 0x71) | DMA Controller Channels 0-3 | | |
| 0x080-  0x08F | DMA Page Register 0-2 (0x81 - 0x83) | DMA Page Register 3 (0x87) | DMA Page Register 4-6 (0x89-0x8B) | DMA Page Register 7 (0x8F) |
| 0x090-  0x09F | System Use | | | |
| 0x0A0-  0x0AF | Interrupt Controller 2 (0xA0-0xA1) | System Use | | |
| 0x0B0-  0x0BF | System Use | | | |
| 0x0C0-  0x0CF | DMA Controller Channels 4-7 (0x0C0-0x0DF), bytes 1-16 | | | |
| 0x0D0-  0x0DF | DMA Controller Channels 4-7 (0x0C0-0x0DF), bytes 16-32 | | | |
| 0x0E0-  0x0EF | System Use | | | |
| 0x0F0-  0x0FF | Floating Point Unit (FPU/NPU/Mah Copprocessor) | | | |
| 0x100-  0x10F | System Use | | | |
| 0x110-  0x11F | System Use | | | |
| 0x120-  0x12F | System Use | | | |
| 0x130-  0x13F | SCSI Host Adapter (0x130-0x14F), bytes 1-16 | | | |
| 0x140-  0x14F | SCSI Host Adapter (0x130-0x14F), bytes 17-32 | | SCSI Host Adapter (0x140-0x15F), bytes 1-16 | |
| 0x150-  0x15F | SCSI Host Adapter (0x140-0x15F), bytes 17-32 | | | |
| 0x160-  0x16F | System Use | | Quaternary IDE Controller, master slave | |
| 0x170-  0x17F | Secondary IDE Controller, Master drive | | System Use | |
| 0x180-  0x18F | System Use | | | |
| 0x190-  0x19F | System Use | | | |
| 0x1A0-  0x1AF | System Use | | | |
| 0x1B0-  0x1BF | System Use | | | |
| 0x1C0- | System Use | | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0x1CF |  | | | |
| 0x1D0-  0x1DF | System Use | | | |
| 0x1E0-  0x1EF | System Use | | Tertiary IDE Controller, master slave | |
| 0x1F0-  0x1FF | Primary IDE Controller, master slave | | System Use | |
| 0x200-  0x20F | Joystick Port | | System Use | |
| 0x210-  0x21F | System Use | | | |
| 0x220-  0x22F | Sound Card | | | |
| Non-NE2000 Network Card | System Use | | |
| 0x230-  0x23F | SCSI Host Adapter (0x220-0x23F), bytes 17-32) | | | |
| 0x240-  0x24F | Sound Card | | | |
| Non-NE2000 Network Card | System Use | | |
| NE2000 Network Card (0x240-0x25F) Bytes 1-16 | | | |
| 0x250-  0x25F | NE2000 Network Card (0x240-0x25F) Bytes 17-32 | | | |
| 0x260-  0x26F | Sound Card | | | |
| Non-NE2000 Network Card | System Use | | |
| NE2000 Network Card (0x240-0x27F) Bytes 1-16 | | | |
| 0x270-  0x27F | System Use | Plug and Play System Devices | LPT2 - Second Parallel Port | |
| System Use | | LPT3 - Third Parallel Port (Monochrome Systems) | |
| NE2000 Network Card (0x260-0x27F) Bytes 17-32 | | | |
| 0x280-  0x28F | Sound Card | | | |
| Non NE2000 Network Card | System Use | | |
| NE2000 Network Card (0x280-0x29F) Bytes 1-16 | | | |
| 0x290-  0x29F | NE2000 Network Card (0x280-0x29F) Bytes 17-32 | | | |
| 0x2A0-  0x2AF | Non NE2000 Network Card | System Use | | |
| NE2000 Network Card (0x280-0x29F) Bytes 1-16 | | | |
| 0x2B0-  0x2BF | NE2000 Network Card (0x280-0x29F) Bytes 17-32 | | | |
| 0x2C0-  0x2CF | System Use | | | |
| 0x2D0-  0x2DF | System Use | | | |
| 0x2E0-  0x2EF | System Use | | COM4 - Fourth Serial Port | |
| 0x2F0-  0x2FF | System Use | | COM2 - Second Serial Port | |
| 0x300-  0x30F | Sound Card / MIDI Port | System Use | | |
| Non NE2000 Network Card | System Use | | |
| NE2000 Network Card (0x300-0x31F) Bytes 1-16 | | | |
| 0x310-  0x31F | NE2000 Network Card (0x300-0x32F) Bytes 17-32 | | | |
| 0x320-  0x32F | Sound Card / MIDI Port (0x330, 0x331) | System Use | | |
| NE2000 Network Card (0x300-0x31F) Bytes 17-32 | | | |
| SCSI Host Adapter (0x330-0x34F) Bytes 1-16 | | | |
| 0x330-  0x33F | Sound Card / MIDI Port | System Use | | |
| Non NE2000 Network Card | System Use | | |
| NE2000 Network Card (0x300-0x31F) Bytes 1-16 | | | |
| 0x340-  0x34F | SCSI Host Adapter (0x330-0x34F) Bytes 17-32 | | | |
| SCSI Host Adapter (0x340-0x35F) Bytes 1-16 | | | |
| Non NE2000 Network Card | System Use | | |
| NE2000 Network Card (0x340-0x35F) Bytes 1-16 | | | |
| 0x350F- | SCSI Host Adapter (0x340-0x35F) Bytes 17-32 | | | |
|  | NE2000 Network Card (0x300-0x31F) Bytes 1-16 | | | |
| 0x360-  0x36F | Tape Accelerator Card (0x360) | System Use | | Quaternary IDE Controller (Slave Drive) (0x36E-0x36F) |
| Non NE2000 Network Card | System Use | | |
| NE2000 Network Card (0x300-0x31F) Bytes 1-16 | | | |
| 0x370- | Tape Accelerator Card | Secondary IDE Controller (Slave | LPT1 - First Parallel Port (Color systems) | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0x37F | (0x370) | Drive) |  | |
| System Use | | LPT2 - Second Parallel Port (Monochrome Systems) | |
| NE2000 Network Card (0x360-0x37F) Bytes 1-16 | | | |
| 0x380-  0x38F | System Use | | Sound Card (FM Synthesizer) | System Use |
| 0x390-  0x39F | System Use | | | |
| 0x3A0-  0x3AF | System Use | | | |
| 0x3B0-  0x3BF | VGA/Monochrome Video | | | LPT1 - First Parallel Port (Monochrome Systems) |
| 0x3C0-  0x3CF | VGA/CGA Video | | | |
| 0x3D0-  0x3DF | VGA/CGA Video | | | |
| 0x3E0-  0x3EF | Tape Accelerator Card (0x370) | System Use | COM3 - Third Serial Port | |
| System Use | | | Tertiary IDE Controller (Slave Drive) (0x3EE-0x3EF) |
| 0x3F0-  0x3FF | Floppy Disk Controller | | COM1 - First Serial Port | |
| Tape Accelerator Card (0x3F0) | Primary IDE Controller (Slave Drive) (0x3F6-0x3F7) | System Use | |

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トレジスタなど、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、（チュートリアルの最初に）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すよね？

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...。

|  |  |  |
| --- | --- | --- |
| WaitLoop: in | al, 64h | ; Get status register value |
| and | al, 10b | ; Test bit 1 of status register |
| jz | WaitLoop | ; If status register bit not set, no data is in buffer |
| in | al, 60h | ; Its set--Get the byte from the buffer (Port 0x60), and store it |

これこそが、ハードウェア・プログラミングとデバイス・ドライバーの基本です。

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ーです。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・コントローラーは、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。割り込みを書くためには、入出力などのハードウェアのルーチンと一緒に、ドライバを書かなければなりません。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・プロセッサー

特別な指示

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) Instructions** | |
| Instruction | Description |
| LGDT | Loads an address of a GDT into GDTR |
| LLDT | Loads an address of a LDT into LDTR |
| LTR | Loads a Task Register into TR |
| MOV *Control Register* | Copy data and store in Control Registers |
| LMSW | Load a new Machine Status WORD |
| CLTS | Clear Task Switch Flag in Control Register CR0 |
| MOV *Debug Register* | Copy data and store in debug registers |
| INVD | Invalidate Cache without writeback |
| INVLPG | Invalidate TLB Entry |
| WBINVD | Invalidate Cache with writeback |
| HLT | Halt Processor |
| RDMSR | Read Model Specific Registers (MSR) |
| WRMSR | Write Model Specific Registers (MSR) |
| RDPMC | Read Performance Monitoring Counter |
| RDTSC | Read time Stamp Counter |

カーネルモードのアクセス権（リング0）を持たない他のプログラムが上記の命令を実行すると、一般保護フォルト、またはトリプルフォルトが発生します。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。

80x86レジスタ

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す。

x86ファミリーには、次のようなレジスタがあります。rax (eax(ax/ah/al)), rbx (ebx(bx/bh/bl)), rcx (ecx(cx/ch/cl)), rdx (edx(dh/dl)), cs,ss,es,ds,fs,gs, rsi (esi (si)), rdi (edi (di)), rbp (ebp (bp)). rsp（esp（sp））、rip（eip（ip））、rflags（eflag（flags））、dr0、dr1、dr2、dr3、dr4、dr5、dr6、dr7、tr1、tr2、tr3、tr4、tr5、tr6、tr7、cr0、cr1、cr2、cr3、cr4、cr8、st,

mm0、mm1、mm2、mm3、mm4、mm5、mm6、mm7、xmm0、xmm1、xmm2、xmm3、xmm4、xmm5、xmm6、xmm7、GDTR、LDTR、IDTR、MSR、TRです。これらのレジスタはすべて、レジスタファイルと呼ばれるプロセッサ内の特別なメモリ領域に格納されています。詳細は「プロセッサ・アーキテクチャ」の項を参照してください。その他のレジスタには、レジスターファイルに含まれていないものもあります。PC、IR、ベクターレジスター、ハードウェアレジスターなどがあります。

これらのレジスターの多くは、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)

その他のレジスタの中には、CPUの内部にあるものもあり、通常の方法ではアクセスできないものもあります。アクセスするためには、プロセッサ自体を再プログラムする必要があります。その中でも特に注目されているのが、IR、ベクターレジスターです。

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。

注：CPUは、私たちがコミュニケーションをとる必要のある通常のデバイスと同じように考えてください。制御レジスタの概念（およびレジスタ自体）は、後に他のデバイスと通信する際に重要になります。

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ーが存在する可能性があります。もしご存知でしたら、追加できるようにお知らせください。）

汎用レジスター

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。

EAX - アキュムレータ・レジスタ。主な用途：数学計算

EBX - ベースアドレスレジスター。主な用途：ベースアドレスを介してメモリに間接的にアクセスする。

ECX - カウンターレジスター。主な用途：カウントやルーピングに使用します。

EDX - データ・レジスタ。主な目的は、データを保存することです。そんなところでしょうか。）

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。高次ワードと低次ワードです。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。64ビットプロセッサでは、これらのレジスタは64ビット幅で、RAX、RBX、RCX、RDXと名付けられています。下位32ビットは、32ビットのEAXレジスタです。

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。

例えば、RAXでは、次のようになります。

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

| | |

+ +

| | | |

+ +

| | |

| +--------EAX lower 32 bits--------------| -- Available only on 32 bit processors.

|

|

|------------------ RAX Complete 64 bits----------------------| -- Available only on 64 bit processors.

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 - Segment address of code segment DS - Segment address of data segment ES - Segment address of extra segment SS - Segment address of stack segment FS - Far Segment address

GS - General Purpose Register

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 - Source Index

DI - Destination Index BP - Base Pointer

SP - Stack Pointer

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ビットの大きさで、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ビットの大きさで、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ビットの大きさで、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。

+---------- EFLAGS (32 Bits) +

| |

|-- FLAGS (16 bits)- |

| | |

==================================================================== ＜レジスタビット

| |

+ RFLAGS (64 Bits) +

| |

Bit 0 Bit 63

|  |  |  |
| --- | --- | --- |
| **FLAGS Register Status Bits** | | |
| Bit Number | Abbrivation | Description |
| 0 | CF | Carry Flag - Status bit |
| 1 |  | Reserved |
| PF | CF | Parity Flag |
| 3 |  | Reserved |
| 4 | AF | Adjust Flag - Status bit |
| 5 |  | Reserved |
| 6 | ZF | Zero Flag - Status bit |
| 7 | SF | Sign Flag - Status bit |
| 8 | TF | Trap Flag (Single Step) - System Flag |
| 9 | IF | Interrupt Enabled Flag - System Flag |
| 10 | DF | Direction Flag - Control Flag |
| 11 | OF | Overflow Flag - Status bit |
| 12-13 | IOPL | I/O Priviledge Level (286+ Only) - Control Flag |
| 14 | NT | Nested Task Flag (286+ Only) - Control Flag |
| 15 |  | Reserved |
| **EFLAGS Register Status Bits** | | |
| Bit Number | Abbrivation | Description |
| 16 | RF | Resume Flag (386+ Only) - Control Flag |
| 17 | VM | v8086 Mode Flag (386+ Only) - Control Flag |
| 18 | AC | Alignment Check (486SX+ Only) - Control Flag |
| 19 | VIF | Virtual Interrupt Flag (Pentium+ Only) - Control Flag |
| 20 | VIP | Virtual Interrupt Pending (Pentium+ Only) - Control Flag |
| 21 | ID | Identification (Pentium+ Only) - Control Flag |
| 22-31 |  | Reserved |
| **RFLAGS Register Status Bits** | | |
| Bit Number | Abbrivation | Description |
| 32-63 |  | Reserved |

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。

For most operations, we only need to use the FLAGS register. Notice that the last 32 bits of the RFLAGS register is nill, null, non existant, there for viewing pleasure. So, um... yeah. For speed puposes, of course, but alot of bytes being wasted yeah.

この表はサイズが大きいので、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。

ビークポイントレジスター

レジスタDR0，DR1，DR2，DR3には，ブレー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。これらのブレークポイント条件は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** - Enable the four debug registers (See below)

**Bit 8...14** - ?

**Bit 15...23** - When the breakpoints will trigger. Each 2 bits represents a single Debug Register. This can be one of the following:

1. - Break on execution
2. - Break on data write
3. - Break on IO read or write. No hardware currently supports this.
4. - Break on data read or write

**Bit 24...31** - Defines how large of memory to watch. Each 2 bits represents a single Debug Register. This can be one of the following:

**00** - One bytes **01** - Two bytes **10** - Eight bytes **11** - Four bytes

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:** Enable local DR0 register

**Bit 1:** Enable global DR0 register

**Bit 2:** Enable local DR1 register

**Bit 3:** Enable global DR1 register

**Bit 4:** Enable local DR2 register

**Bit 5:** Enable global DR2 register

**Bit 6:** Enable local DR3 register

**Bit 7:** Enable global DR3 register

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こったかを判断するためにデバッガで使用されます。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** - Read from MSR

**WRMSR** - Write from MSR

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。

|  |  |  |
| --- | --- | --- |
| **Model Specific Registers (MSRs)** | | |
| Register Address | Register Name | IA-32 Processor Family |
| 0x0 | IA32\_PS\_MC\_ADDR | Pentium Processors |
| 0x1 | IA32\_PS\_MC\_TYPE | Pentium 4 Processors |
| 0x6 | IA32\_PS\_MONITOR\_FILTER\_SIZE | Pentium Processors |
| 0x10 | IA32\_TIME\_STAMP\_COUNTER | Pentium Processors |
| 0x17 | IA32\_PLATFORM\_ID | P6 Processors |
| 0x1B | IA32\_APIC\_BASE | P6 Processors |
| 0x3A | IA32\_FEATURE\_CONTROL | Pentium 4 / Processor 673 |
| 0x79 | IA32\_BIOS\_UPDT\_TRIG | P6 Processors |
| 0x8B | IA32\_BIOS\_SIGN\_ID | P6 Processors |
| 0x9B | IA32\_SMM\_MONITOR\_CTL | Pentium 4 / Processor 672 |
| 0xC1 | IA32\_PMC0 | Intel Core Duo |
| 0xC2 | IA32\_PMC1 | Intel Core Duo |
| 0xE7 | IA32\_MPERF | Intel Core Duo |
| 0xE8 | IA32\_APERF | Intel Core Duo |
| 0xFE | IA32\_MTRRCAP | P6 Processors |
| 0x174 | IA32\_SYSENTER\_CS | P6 Processors |
| 0x175 | IA32\_SYSENTER\_ESP | P6 Processors |
| 0x176 | IA32\_SYSENTER\_IP | P6 Processors |

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A ppendex 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、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）。

; Now EDX:EAX contains the lower and upper 32 bits of the 64 bit register

; Register 0x174: IA32\_SYSENTER\_CS

; Read in the MSR

mov cx, 0x174 rdmsr

; This reads from the IA32\_SYSENTER\_CS MSR

かっこいいでしょう？

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。

; Register 0x174: IA32\_SYSENTER\_CS

; Write EDX:EAX into the MSR

mov cx, 0x174 wrmsr

; This writes to the IA32\_SYSENTER\_CS MSR

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。

**Bit 0 (PE) : Puts the system into protected mode**

**Bit 1 (MP) : Monitor Coprocessor Flag** This controls the operation of the **WAIT** instruction. **Bit 2 (EM) : Emulate Flag**. When set, **coprocessor instructions will generate an exception Bit 3 (TS) : Task Switched Flag** This will be set when the processor switches to another **task**. **Bit 4 (ET) : ExtensionType Flag. This tells us what type of coprocesor is installed.**

1. - 80287 is installed
2. - 80387 is installed.

**Bit 5 (NE):** Numeric Error

1. - Enable standard error reporting
2. - Enable internal x87 FPU error reporting

**Bits 6-15 :** Unused

**Bit 16 (WP):** Write Protect

**Bit 17:** Unused

**Bit 18 (AM):** Alignment Mask

1. - Alignment Check Disable
2. - Alignment Check Enabled (Also requires AC flag set in EFLAGS and ring 3)

**Bits 19-28:** Unused

**Bit 29 (NW):** Not Write-Through

**Bit 30 (CD):** Cache Disable

**Bit 31 (PG) : Enables Memory Paging.**

1. - Disable
2. - Enabled and use CR3 register

うわぁ...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。

; get value in CR0

; set bit 0--enter protected mode

; Bit 0 now set, we are in 32 bit mode!

ax, cr0 ax, 1 cr0, ax

mov or mov

すげええええええええ 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ールトになります。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ールトになります。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ブレー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) :** Enables Virtual 8086 Mode Extensions

**Bit 1 (PVI) :** Enables Protected Mode Virtual Interrupts

**Bit 2 (TSD) :** Time Stamp Enable

1. - RDTSC instruction can be used in any privilege level 1 - RDTSC instruction can only be used in ring 0

**Bit 3 (DE) :** Enable Debugging Extensions

**Bit 4 (PSE) :** Page Size Extension 0 - Page size is 4KB

1. - Page size is 4MB. With PAE, it is 2MB.

**Bit 5 (PAE) :** Physical Address Extension **Bits 6 (MCE) :** Machine Check Exception **Bits 7 (PGE) :** Page Global Enable

**Bits 8 (PCE) :** Performance Monitoring Counter Enable

0 - RDPMC instruction can be used in any privilege level 1 - RDPMC instruction can only be used in ring 0

**Bits 9 (OSFXSR) :** OS Support for FXSAVE and FXSTOR instructions (SSE) **Bits 10 (OSXMMEXCPT) :** OS Support for unmasked SIMD FPU exceptions **Bits 11-12 :** Unused

**Bits 13 (VMXE) :** VMX Enable

CR8コントロールレジスタ

タスクプライオリティーレジスタ（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。これらのレジスターは

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、これらのレジスター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。ここまでで「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。

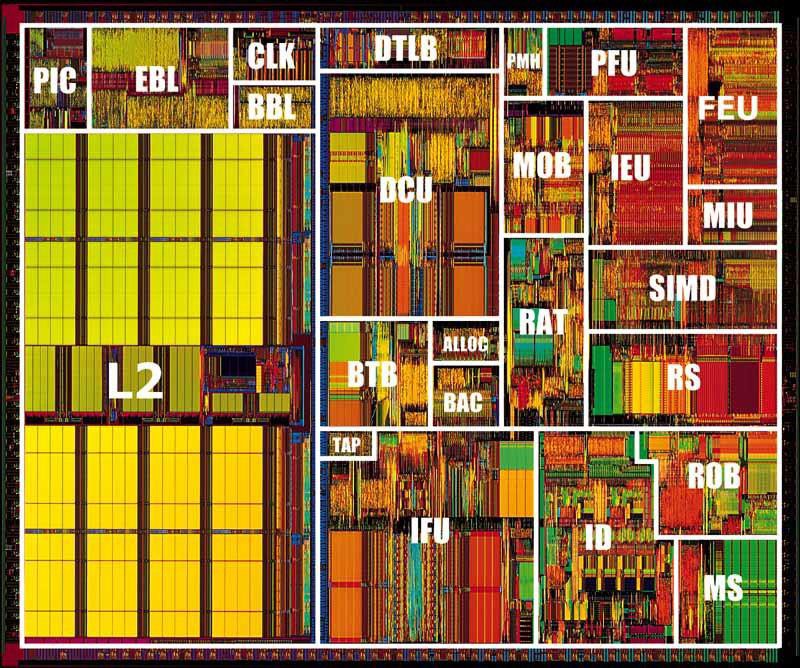
注：このセクションはかなり専門的です。すべてを理解する必要はありませんので、すべてを理解できなくても心配はありません。私がこのセクションを設けたのは、コンピュータシステムに必要な主要コンポーネント、そして私たちのコードを実行する責任のあるコンポーネントを完全なものにするためです。与えられたコードを

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ーの開発に入ると、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。

プロセッサーを分解する

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。



プロセッサー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:** Level 2 Cache

**CLK:** Clock

**PIC: Programmable Interrupt Controller EBL:** Front Bus Logic

**BBL:** Back Bus Logic

**IEU:** Integer Execution Unit

**FEU:** Floating Point Execution Unit

**MOB:** Memory Order Buffer

**MIU / MMU:** Memory Interface Unit / Memory Management Unit

**DCU:** Data Cach Unit

**IFU:** Instruction Fetch Unit

**ID:** Instruction Decoder

**ROB:** Re-Order Buffer

**MS: Microinstruction** Sequencer

**BTB:** Branch Target Buffer

**BAC:** Branch Allocater Buffer

**RAT:** Register Alias Table

**SIMD:** Packed floating point

**DTLB:** Data TLB

**RS:** Reservation Station

**PMH:** Page Miss Handler

**PFU:** Pre-Fetch Unit

**TAP:** Test Access Port

*I plan on adding to this section.*

### 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＝segment \* 16 + 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＝CS \* 16 + 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。

メモリコントローラが制御する。メモリーコントローラーは、アドレスバスからアドレスをコピーし、パーキュラーRAMチップの正確な位置を計算します。メモリコントローラは、この位置を参照し、データバスにコピーします。これを行うのは、「Read Memory」ラインが

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

プロセッサーは、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、コントローラー内の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（Arithmitic 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さ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ーに依存しています。

### 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。

1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、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ーによって、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ても、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詳細に説明しています。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、これからたくさんのこと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、手順ごとに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と、プログラムがクラッシュしてしまいます。

楽しみですね... :) 次の機会まで。

~マイク

*BrokenThorn Entertainment社。現在、DoEとN eptune Operating Systemを開発中 質問やコメントは？ お気軽にご連絡ください。*

記事をより良くするために、あなたも貢献してみませんか？よろしければ、ぜひご連絡ください。



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

Protected Mode Theory

Protected Mode Addressing

Entering Protected Mode Global Descriptor Table (GDT)

*Ready?*

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  ; -Input/Output routines  ;  ; OS Development Series  ;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  %ifndef STDIO\_INC\_67343546FDCC56AAB872\_INCLUDED  %define STDIO\_INC\_67343546FDCC56AAB872\_INCLUDED  ;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;  ; Puts16 ()  ; -Prints a null terminated string  ; DS=>SI: 0 terminated string  ;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;  bits 16 | | | |
| Puts16: |  |  |  |
|  | pusha |  | ; save registers |
| .Loop1: |  |  |  |
|  | lodsb |  | ; load next byte from string from SI to AL |
|  | or | al, al | ; Does AL=0? |
|  | jz | Puts16Done | ; Yep, null terminator found-bail out |
|  | mov | ah, 0eh | ; Nope-Print the character |
|  | int | 10h | ; invoke BIOS |
|  | jmp | .Loop1 | ; Repeat until null terminator found |
| Puts16Done: |  |  |  |
|  | popa ret |  | ; restore registers  ; we are done, so return |
| %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、ブートローダで使用し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。

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。

**Enable and go into protected mode Retrieve BIOS information**

**Load and execute the kernel**

Provide advance boot options (Such as Safe Mode, for example)

Through configuration files, you can have KRNLDR boot from multiple operating system kernels

**Enable the 20th address line for access up to 4 GB of memory Provide basic interrupt handling**

...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でしょうか。

プロテクトモードの世界

Yippie! いよいよですね。プロテクトモード」という言葉をよく耳にしますし、これまでにも詳しく説明してきました。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）例外を発生させます。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。）

The GDT usually has three descriptors--a **Null descriptor** (Contains all zeros), a **Code Descriptor**, and a **Data Descriptor**.

Okay So, what is a "Descriptor"? For the GDT, a "Descriptor" is an 8 byte QWORD value that describes properties for the descriptor.

形式になっています。

**Bits 56-63:** Bits 24-32 of the base address

**Bit 55:** Granularity

**0:** None

**1:** Limit gets multiplied by 4K

**Bit 54:** Segment type

**0:** 16 bit

**1:** 32 bit

**Bit 53:** Reserved-Should be zero

**Bits 52:** Reserved for OS use

**Bits 48-51:** Bits 16-19 of the segment limit

**Bit 47** Segment is in memory (Used with Virtual Memory)

**Bits 45-46:** Descriptor Privilege Level

**0:** (Ring 0) Highest

**3:** (Ring 3) Lowest

**Bit 44:** Descriptor Bit

**0:** System Descriptor

**1:** Code or Data Descriptor

**Bits 41-43:** Descriptor Type

**Bit 43:** Executable segment

**0:** Data Segment

**1:** Code Segment

**Bit 42:** Expansion direction (Data segments), conforming (Code Segments)

**Bit 41:** Readable and Writable

**0:** Read only (Data Segments); Execute only (Code Segments)

**1:** Read and write (Data Segments); Read and Execute (Code Segments)

**Bit 40:** Access bit (Used with Virtual Memory)

**Bits 16-39:** Bits 0-23 of the Base Address

**Bits 0-15:** Bits 0-15 of the Segment Limit

...かなり醜いですよね？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。

; data descriptor

; limit low (Same as code)

; base low

; base middle

; access

; granularity

; base high

; data descriptor:

dw 0FFFFh dw 0

db 0

db 10010010b db 11001111b

db 0

; Because each descriptor is 8 bytes in size, the Data descritpor is at offset 0x10 from

; the beginning of the GDT, or 16 (decimal) bytes from start.

; code descriptor. Right after null descriptor

; limit low

; base low

; base middle

; access

; granularity

; base high

; code descriptor:

dw 0FFFFh dw 0

db 0

db 10011010b db 11001111b

db 0

; Notice that each descriptor is exactally 8 bytes in size. THIS IS IMPORTANT.

; Because of this, the code descriptor has offset 0x8.

; null descriptor--just fill 8 bytes with zero

; null descriptor

dd 0

dd 0

; This is the beginning of the GDT. Because of this, its offset is 0.

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。

何が起こっているのか、それぞれのビットに分解してみましょう。ヌルデスクリプタはすべてゼロなので、残りの2つに注目します。コードセレクターを分解する

もう一度見てみましょう。

; code descriptor:

dw 0FFFFh dw 0

db 0

db 10011010b db 11001111b

db 0

; code descriptor. Right after null descriptor

; limit low

; base low

; base middle

; access

; granularity

; base high

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、コードセレクターは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

; access

**Bit 0 (Bit 40 in GDT):** Access bit (Used with Virtual Memory). Because we don't use virtual memory (Yet, anyway), we will ignore it. Hence, it is 0

**Bit 1 (Bit 41 in GDT):** is the readable/writable bit. Its set (for code selector), so we can read and execute data in the segment (From 0x0 through 0xFFFF) as code

**Bit 2 (Bit 42 in GDT):** is the "expansion direction" bit. We will look more at this later. For now, ignore it.

**Bit 3 (Bit 43 in GDT):** tells the processor this is a code or data descriptor. (It is set, so we have a code descriptor)

**Bit 4 (Bit 44 in GDT):** Represents this as a "system" or "code/data" descriptor. This is a code selector, so the bit is set to 1.

**Bits 5-6 (Bits 45-46 in GDT):** is the privilege level (i.e., Ring 0 or Ring 3). We are in ring 0, so both bits are 0.

**Bit 7 (Bit 47 in GDT):** Used to indicate the segment is in memory (Used with virtual memory). Set to zero for now, since we are not using virtual memory yet

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。

; granularity

; base high

db 11001111b

db 0

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。

**Bit 0-3 (Bits 48-51 in GDT):** Represents bits 16-19 of the segment limit. So, lessee... 1111b is equal to 0xf. Remember that, in the first two bytes if this descriptor, we set 0xffff as the first 15 bites. Grouping the low and high bits, **it means we can access up to 0xFFFFF**. Cool? It gets better... By enabling the 20th address line, we can access **up to 4 GB of memory** using this descriptor. We will look closer at this later...

**Bit 4 (Bit 52 in GDT):** Reserved for our OS's use--we could do whatever we want here. Its set to 0.

**Bit 5 (Bit 53 in GDT):** Reserved for something. Future options, maybe? Who knows. Set to 0.

**Bit 6 (Bit 54 in GDT):** is the segment type (16 or 32 bit). Lessee. we want 32 bits, don't we? After all-we are building a 32 bit

OS! So, yeah Set to 1.

**Bit 7 (Bit 55 in GDT):** Granularity. By setting to 1, each segment will be bounded by 4KB. The last byte is bits 24-32 of the base (Starting) address which, of course is 0.

That's all there is to it!

データ記述子

それでは、作成したGDTに戻って、コードセレクターとデータセレ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。コードセレクターの場合はセットされ、データセレ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。

toc:

dw end\_of\_gdt - gdt\_data - 1 ; limit (Size of GDT) dd gdt\_data ; base of GDT

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 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。そうしないと、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。

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を使用しています。GDTをGDTRレジスタにロードするには、特別な命令...LGDT（Load GDT）が必要です。使い方はとても簡単です。

lgdt [toc]

; load GDT into GDTR

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ので、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、コードセレ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

;...Other descriptors begin at offset 0x18. Remember that each descriptor is 8 bytes in size?

; Add other descriptors for Ring 3 applications, stack, whatever here...

end\_of\_gdt:

toc:

dw end\_of\_gdt - gdt\_data - 1 ; limit (Size of GDT) dd gdt\_data ; base of GDT

; data descriptor

; limit low (Same as code)

; base low

; base middle

; access

; granularity

; base high

; gdt data:

dw 0FFFFh dw 0

db 0

db 10010010b db 11001111b

db 0

; Offset 16 bytes (0x10) from start of GDT. Descriptor code therfore is 0x10.

; code descriptor

; limit low

; base low

; base middle

; access

; granularity

; base high

; gdt code:

dw 0FFFFh dw 0

db 0

db 10011010b db 11001111b

db 0

; Offset 0x8 bytes from start of GDT: Descriptor code therfore is 8

; null descriptor

; Offset 0 in GDT: Descriptor code=0

gdt\_data:

dd 0

dd 0

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。

|  |  |  |
| --- | --- | --- |
| cli |  | ; make sure to clear interrupts first! |
| lgdt | [toc] | ; load GDT into GDTR |

sti

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 Routines

OS Development Series

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

%ifndef GDT\_INC\_67343546FDCC56AAB872\_INCLUDED

%define GDT\_INC\_67343546FDCC56AAB872\_INCLUDED

bits 16

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; InstallGDT()

; - Install our GDT

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

InstallGDT:

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Global Descriptor Table (GDT)

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

gdt\_data:

dd 0

dd 0

; null descriptor

; gdt code:

dw 0FFFFh dw 0

db 0

db 10011010b db 11001111b

db 0

; code descriptor

; limit low

; base low

; base middle

; access

; granularity

; base high

; gdt data:

dw 0FFFFh dw 0

db 0

db 10010010b db 11001111b

db 0

; data descriptor

; limit low (Same as code)

; base low

; base middle

; access

; granularity

; base high

end\_of\_gdt:

toc:

dw end\_of\_gdt - gdt\_data - 1 ; limit (Size of GDT) dd gdt\_data ; base of GDT

%endif ; GDT\_INC\_67343546FDCC56AAB872\_INCLUDED

|  |  |  |
| --- | --- | --- |
| cli |  | ; clear interrupts |
| pusha |  | ; save registers |
| lgdt | [toc] | ; load GDT into GDTR |
| sti |  | ; enable interrupts |
| popa |  | ; restore registers |
| ret |  | ; All done! |

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...。

**Bit 0 (PE) : Puts the system into protected mode**

**Bit 1 (MP) : Monitor Coprocessor Flag** This controls the operation of the **WAIT** instruction.

**Bit 2 (EM) : Emulate Flag**. When set, **coprocessor instructions will generate an exception**

**Bit 3 (TS) : Task Switched Flag** This will be set when the processor switches to another **task**.

**Bit 4 (ET) : ExtensionType Flag. This tells us what type of coprocessor is installed.**

1. - 80287 is installed
2. - 80387 is installed.

**Bit 5 :** Unused.

**Bit 6 (PG) : Enables Memory Paging.**

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ビットの状態で実行を継続します。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。

|  |  |  |
| --- | --- | --- |
| mov | eax, cr0 | ; set bit 0 in CR0-go to pmode |
| or | eax, 1 |  |
| mov | cr0, eax |  |

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:Stage3

; far jump to fix CS. Remember that the code selector is 0x8!

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 mov mov | ax,  ds,  ss, | 0x10  ax ax | ; | set | data | segments | to | data | selector | (0x10) |
| mov | es, | ax | | | | | | | | |

データ記述子は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。

bits 16

; Remember the memory map-- 0x500 through 0x7bff is unused above the BIOS data area.

; We are loaded at 0x500 (0x50:0)

org 0x500

jmp

main

; go to start

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Preprocessor directives

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

%include "stdio.inc"

%include "Gdt.inc"

; basic i/o routines

; Gdt routines

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Data Section

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

LoadingMsg db "Preparing to load operating system...", 0x0D, 0x0A, 0x00

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

;

;

;

;

;

;

STAGE 2 ENTRY POINT

-Store BIOS information

-Load Kernel

-Install GDT; go into protected mode (pmode)

-Jump to Stage 3

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* main:

;-------------------------------;

; Setup segments and stack ;

;-------------------------------;

;-------------------------------;

; Print loading message ;

;-------------------------------;

mov si, LoadingMsg

call Puts16

;-------------------------------;

|  |  |  |  |
| --- | --- | --- | --- |
| cli |  | ; | clear interrupts |
| xor | ax, ax | ; | null segments |
| mov | ds, ax |  |  |
| mov | es, ax |  |  |
| mov | ax, 0x9000 | ; | stack begins at 0x9000-0xffff |
| mov | ss, ax |  |  |
| mov | sp, 0xFFFF |  |  |
| sti |  | ; | enable interrupts |

; Install our GDT

;

;-------------------------------;

call InstallGDT

; install our GDT

;-------------------------------;

; Go into pmode ;

;-------------------------------;

cli mov or mov

eax, cr0 eax, 1 cr0, eax

; clear interrupts

; set bit 0 in cr0--enter pmode

jmp

08h:Stage3

; far jump to fix CS. Remember that the code selector is 0x8!

; Note: Do NOT re-enable interrupts! Doing so will triple fault!

; We will fix this in Stage 3.

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; ENTRY POINT FOR STAGE 3

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

bits 32

; Welcome to the 32 bit world!

Stage3:

;-------------------------------;

; Set registers ;

;-------------------------------;

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Stop execution

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* STOP:

cli hlt

|  |  |  |
| --- | --- | --- |
| mov | ax, 0x10 | ; set data segments to data selector (0x10) |
| mov | ds, ax |  |
| mov | ss, ax |  |
| mov | es, ax |  |
| mov | esp, 90000h | ; stack begins from 90000h |

結論

私は興奮していますが、あなたはどうですか？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。かっこいいでしょう？

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。

Direct Hardware Programming - Theory

Direct Hardware Programming and Controllers

Keyboard Controller Programming - Basics

Enabling A20

Pizza :)

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ので、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。ボークスは、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。



[0x000ffff0] f000:fff0 (unk. ctxt): jmp f000:e05b

< bochs:1> \_

; ea5be000f0

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。

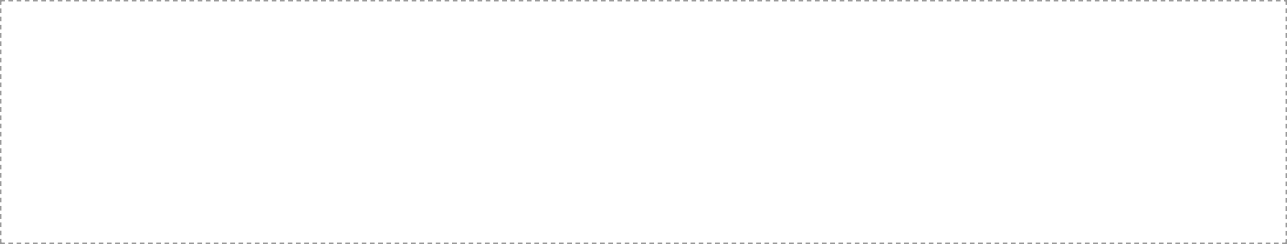
1行目が重要な行です。現在の命令、絶対アドレス、seg:offsetアドレスを教えてくれます。また、機械語のオペレーションコード（Opcode）に相当するものもわかります。

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（07c00:0）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:fff0 (unk. ctxt): jmp f000:e05b

< bochs:1> b 0x7c00 // Sets the breakpoint to 0x7c00:0

< bochs:2> c // Continue executon

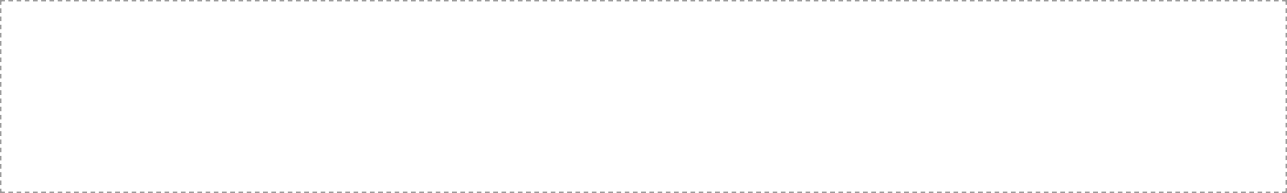
< 0> Breakpoint 1, 0x7c00 in ?? < > // Our breakpoint is hit Next at t=834339

// We are now at our bootloaders first instruction

< 0> [0x00007c00] 0000:7c00 (unk. ctxt): jmp 7cb5 ; e9b200/DIV>

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> Breakpoint 2, 0x500 in ?? < > Next at t=934606

<0> [0x000000500] 0050:0000 (ink. ctxt): jmp 00a0

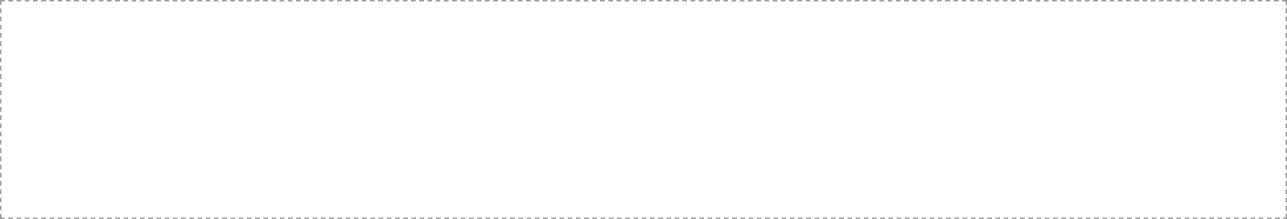
< bochs:5> \_

; e99d00

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。かっこいいでしょう？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

< bochs:7> s Next at t=934608

<0> [0x0000005a1] 0050:00a1 (ink. ctxt): xor AX, AX

< bochs:8> \_

; fa

; 31c0

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。

"ダイレクト・ハードウェア・プログラミング "とは、簡単に言えば、個々のチップと直接通信して（コントロールして）使用することです。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アーキテクチャにおける一般的なポート・マッピングの膨大な表を示しました。

プロセッサーが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コントローラはそのアドレスを受け取る。

I/Oコントローラは、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

これらの仕組みを理解するために、コントローラを見てみましょう。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、電流が流れるようになっている。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、いつでも電気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

; get byte from 8042 microcontroller input register

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可能になります。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ー「インテル80286」を採用していたが、このマイクロプロセッサーは、従来の

「x86」マイクロプロセッサー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を行う際に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、出力ラインデータを書き込むことで、マイコンにORゲートを設定させ、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いてください。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。

**Bit 0** - Setting to 1 causes a fast reset (Used to switch back to real mode)

**Bit 1** - 0: disable A20; 1: enable A20

**Bit 2** - Manufacturer defined

**Bit 3** - power on password bytes (CMOS bytes 0x38-0x3f or 0x36-0x3f). 0: accessible, 1: inaccessible

**Bits 4-5** - Manufacturer defined

**Bits 6-7** - 00: HDD activity LED off; any other value is "on"

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。

al, 2 ; set bit 2 (enable a20) 0x92, al

mov out

このポートでできることは他にもたくさんあります。

al, 1 ; set bit 1 (fast reset) 0x92, al

mov out

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を無効にするには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

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

戻ります。

CF = 成功したらクリア

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 = clear if success

AH = status (01: keyboard controller is in secure mode; 0x86: function not supported)

AL = current state (00: disabled, 01: enabled)

CX = set to 0xffff is keyboard controller is no ready in 0xc000 read attempts

CF = set on error

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

戻ります。

CF = clear if success

AH = status (01: keyboard controller is in secure mode; 0x86: function not supported)

BX = status.

BX contains a bit pattern:

**Bit 0** - 1 if supported on keyboard controller

**Bit 1** - 1 if supported on bit 1 of I/O port 0x92

**Bits 2-14** - Reserved

**Bit 15** - 1 if additional data is available.

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。この方法では、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。心配しなくても、そんなに悪くはありません。） 時には複雑になることもありますが 。）

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。

|  |  |  |
| --- | --- | --- |
| **Port Mapping** | | |
| Port | Read/Write | Descripton |
| 0x60 | Read | Read Input Buffer |
| 0x60 | Write | Write Output Buffer |
| 0x64 | Read | Read Status Register |
| 0x64 | Write | Send Command to controller |

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。このレジスター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

**Bit 0: Output Buffer Status**

0: Output buffer empty, dont read yet 1: Output buffer full, please read me :)

**Bit 1: Input Buffer Status**

0: Input buffer empty, can be written 1: Input buffer full, dont write yet

**Bit 2:** System flag

0: Set after power on reset

1: Set after successfull completion of the keyboard controllers self-test (Basic Assurance Test, BAT)

**Bit 3:** Command Data

0: Last write to input buffer was data (via port 0x60)

1: Last write to input buffer was a command (via port 0x64)

**Bit 4:** Keyboard Locked 0: Locked

1: Not locked

**Bit 5:** Auxiliary Output buffer full PS/2 Systems:

0: Determins if read from port 0x60 is valid If valid, 0=Keyboard data 1: Mouse data, only if you can read from port 0x60

AT Systems:

0: OK flag

1: Timeout on transmission from keyboard controller to keyboard. **This may indicate no keyboard is present.**

**Bit 6:** Timeout

0: OK flag

1: Timeout PS/2:

AT:

General Timeout

Timeout on transmission from keyboard to keyboard controller. **Possibly parity error (In which case both bits 6 and 7 are set)**

**Bit 7:** Parity error

0: OK flag, no error

1: Parity error with last byte

ご覧のように、ここでは多くのことが行われています。重要な部分は上の太字の部分で、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:

|  |  |  |
| --- | --- | --- |
| in | al,0x64 | ; read status register |
| test | al,2 | ; test bit 2 (Input buffer status) |
| jnz | wait\_input | ; jump if its not 0 (not empty) to continue waiting |

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ポート表を見てみると、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ー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。

|  |  |
| --- | --- |
| **Keyboard Controller Commands** | |
| Keyboard Command | Descripton |
| 0x20 | Read Keyboard Controller Command Byte |
| 0x60 | Write Keyboard Controller Command Byte |
| 0xAA | Self Test |
| 0xAB | Interface Test |
| 0xAD | Disable Keyboard |
| 0xAE | Enable Keyboard |
| 0xC0 | Read Input Port |
| 0xD0 | Read Output Port |
| 0xD1 | Write Output Port |
| 0xDD | Enable A20 Address Line |
| 0xDF | Disable A20 Address Line |
| 0xE0 | Read Test Inputs |
| 0xFE | System Reset |
| Mouse Command | Descripton |
| 0xA7 | Disable Mouse Port |
| 0xA8 | Enable Mouse Port |
| 0xA9 | Test Mouse Port |
| 0xD4 | Write to mouse |

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ーで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。

; Method 3.1: Enables A20 through keyboard controller mov al, 0xdd ; command 0xdd: enable a20

out 0x64, al ; send command to controller

すべてのキーボードコントローラーがこの機能に対応してい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。）

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ー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。）

; read output port into al mov al,0xD0

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。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。*

**Bit 0: System Reset**

0: Reset computer

1: Normal operation

**Bit 1: A20**

0: Disabled

1: Enabled

**Bit 2-3:** Undefined

**Bit 4:** Input buffer full

**Bit 5:** Output Buffer Empty

**Bit 6:** Keyboard Clock

0: High-Z

1: Pull Clock Low

**Bit 6:** Keyboard Data

0: High-Z

1: Pull Data Low

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。他のビットに影響が出ないように、この値をORしてビットを設定してください。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ーに入ることになります。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。

|  |  |  |
| --- | --- | --- |
| ; send read output port command mov al,0xD0  out 0x64,al  call wait\_output  ; read input buffer and store on stack. This is the data read from the output port in al,0x60  push eax  call wait\_input  ; send write output port command mov al,0xD1  out 0x64,al  call wait\_input  ; pop the output port data from stack and set bit 1 (A20) to enable | | |
| pop | eax |  |
| or | al,2 | // 2 = 10 binary |
| out | 0x60,al | // write the data to the output port. This is done through the output buffer |

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

コントローラー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。

コントローラーの誤動作

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（Hault 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、これらの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。(コントローラー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ーは、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ープロセッサーに通知することができ、その場合、プロセッサーは例外を通知し、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

コントローラーに問題がある場合、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。

|  |  |  |
| --- | --- | --- |
| mov | al, 0x00 | ; some random command |
| out | 0x64, al | ; try to send command to controller |

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 ] unsupported io write to keyboard port 64, value = 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。

**Demo**

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ーをプログラミングして、そのためのドライバーを書くためのチュートリアル全体を用意するつもりです。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N eptune Operating Systemを開発中 質問やコメントは？お気軽にご連絡ください。*

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の第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の第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ッシングマップとデフォルト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を始めるまで、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プを覚えていますか?) 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コントローラー

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ッシング、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ー、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信される。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の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、ビデオドライバー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することができ、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、下位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のオリジナル）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0xA0000～0xBFFFFにマッピングされています。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され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ーは、VGAポートを通じてモニター内のCRTコントローラー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なので、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

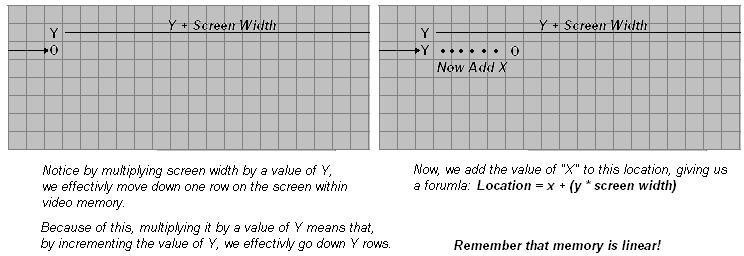
; video memory

|  |  |  |
| --- | --- | --- |
| mov | edi, VIDMEM | ; get pointer to video memory |
| mov | [edi], 'A' | ; print character 'A' |
| mov | [edi+1], 0x7 | ; character attribute |

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。かっ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。

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。

address = x + (y \* screen width) address = 5 + (5 \* 80)

address = 5 + 400

address = 405

This means, from the start of video memory, location 5,5 is 405 bytes away. So, add this to the base address of video memory:

0xB8000 + 405 = 0xB8195

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るのです。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。

|  |  |  |
| --- | --- | --- |
| \_CurX db 0  \_CurY db 0 |  | ; current x/y location |
| %define | VIDMEM 0xB8000 | ; video memory |
| %define  %define  %define | COLS 80  LINES 25  CHAR\_ATTRIB 14 | ; width and height of screen  ; character attribute (White text on black background) |

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行となる。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 ちょ

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、

1. - Black
2. - Blue
3. - Green
4. - Cyan
5. - Red
6. - Magenta
7. - Brown
8. - Light Gray

**8** - Dark Gray

**9** - Light Blue

**10** - Light Green

**11** - Light Cyan

**12** - Light Red

**13** - Light Magenta

**14** - Light Brown

**15** - White

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。

**Bits 0 - 2:** Foreground color

Bit 0: Red

Bit 1: Green

Bit 2: Blue

**Bit 3:** Foreground Intensity

**Bits 4 - 6:** Background color

Bit 4: Red

Bit 5: Green

Bit 6: Blue

**Bit 7:** Blinking or background intensity

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に書き換えます。アセンブリーでどのように行われているかを示すことで、アセンブリー言語の関係を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。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| bits 32 |  |  | | |
| %define | VIDMEM 0xB8000 | ; video memory | | |
| %define  %define  %define | COLS 80  LINES 25  CHAR\_ATTRIB 14 |  | ; width and height of screen  ; character attribute (White text on black | background) |
| \_CurX db 0  \_CurY db 0 |  | ; current x/y location | | |
| ;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;  ; Putch32 ()  ; - Prints a character to screen  ; BL => Character to print  ;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;  Putch32: | | | | |
| pusha mov | edi, VIDMEM | ;  ; | save registers  get pointer to video memory |  |

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、実質的に行の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)に書き込めばいい。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ではありません。

;-------------------------------;

; Get current position ;

;-------------------------------;

xor

eax, eax

; clear eax

;--------------------------------

; Remember: currentPos = x + y \* COLS! x and y are in \_CurX and \_CurY.

; Because there are two bytes per character, COLS=number of characters in a line.

; We have to multiply this by 2 to get number of bytes per line. This is the screen width,

; so multiply screen with \* \_CurY to get current line

;--------------------------------

|  |  |  |
| --- | --- | --- |
| mov | ecx, COLS\*2 | ; Mode 7 has 2 bytes per char, so its COLS\*2 bytes per line |
| mov | al, byte [\_CurY] | ; get y pos |
| mul | ecx | ; multiply y\*COLS |
| push | eax | ; save eax--the multiplication |

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。

;--------------------------------

; Now y \* screen width is in eax. Now, just add \_CurX. But, again remember that \_CurX is relative

; to the current character count, not byte count. Because there are two bytes per character, we

; have to multiply \_CurX by 2 first, then add it to our screen width \* y.

;--------------------------------

|  |  |  |
| --- | --- | --- |
| mov | al, byte [\_CurX] | ; multiply \_CurX by 2 because it is 2 bytes per char |
| mov | cl, 2 |  |
| mul | cl |  |
| pop | ecx | ; pop y\*COLS result |
| add | 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。

; add it to the base address

ecx, ecx edi, eax

xor add

;

; Now eax contains the offset address to draw the character at, so just add it to the base address

; of video memory (Stored in edi)

;

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。

;-------------------------------;

; Watch for new line ;

;-------------------------------;

;-------------------------------;

; Print a character ;

;-------------------------------;

;-------------------------------;

; Update next position ;

;-------------------------------;

|  |  |  |  |
| --- | --- | --- | --- |
| cmp | bl, 0x0A | ; | is it a newline character? |
| je | .Row | ; | yep--go to next row |

|  |  |  |  |
| --- | --- | --- | --- |
| mov | dl, bl | ; | Get character |
| mov | dh, CHAR\_ATTRIB | ; | the character attribute |
| mov | word [edi], dx | ; | write to video display |

|  |  |  |  |
| --- | --- | --- | --- |
| inc | byte [\_CurX] | ; | go to next character |
| cmp | [\_CurX], COLS | ; | are we at the end of the line? |
| je | .Row | ; | yep-go to next row |
| 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。

; restore registers and return

popa ret

.done:

; ;

; Restore registers & return ;

; ;

; go back to col 0

; go to next row

byte [\_CurX], 0 byte [\_CurY]

mov inc

.Row:

;

;

;

;

;

; Go to next row

### 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:

; ;

; Store registers ;

; ;

|  |  |  |
| --- | --- | --- |
| pusha |  | ; save registers |
| push | ebx | ; copy the string address |
| pop | 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ので、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。

.loop:

;-------------------------------;

; Get character ;

;-------------------------------;

|  |  |  |
| --- | --- | --- |
| mov | bl, byte [edi] | ; get next character |
| cmp | bl, 0 | ; is it 0 (Null terminator)? |
| je | .done | ; yep-bail out |

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。

; Nope-print it out

call Putch32

;-------------------------------;

; Print the character ;

;-------------------------------;

...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。

;-------------------------------;

; Go to next character ;

;-------------------------------;

.Next:

inc jmp

edi

.loop

; go to next character

.done:

;-------------------------------;

; Update hardware cursor ;

;-------------------------------;

; Its more efficiant to update the cursor after displaying

; the complete string because direct VGA is slow

|  |  |  |
| --- | --- | --- |
| mov mov call | bh, byte [\_CurY] bl, byte [\_CurX] MovCur | ; get current position  ; update cursor |
| popa ret |  | ; restore registers, and return |

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。その時に全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ましょう。

ポートマッピング

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。

つまり、CRTコントローラにデータを与えるためには、2つの値を書き込まなければならないのです。1つはインデックス・レジスタ（書き込もうとしているデータの種類を含む）に、もう1つはデータ・レジスタ（データを含む）に。難しいことではありませんね :)

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 Microcontroller - Index Register** | |
| Index Offset | CRT Controller Register |
| 0x0 | Horizontal Total |
| 0x1 | Horizontal Display Enable End |
| 0x2 | Start Horizontal Blanking |
| 0x3 | End Horizontal Blanking |
| 0x4 | Start Horizontal Retrace Pulse |
| 0x5 | End Horizontal Retrace |
| 0x6 | Vertical Total |
| 0x7 | Overflow |
| 0x8 | Preset Row Scan |
| 0x9 | Maximum Scan Line |
| 0xA | Cursor Start |
| 0xB | Cursor End |
| 0xC | Start Address High |
| 0xD | Start Address Low |
| 0xE | Cursor Location High |
| 0xF | Cursor Location Low |
| 0x10 | Vertical Retrace Start |
| 0x11 | Vertical Retrace End |
| 0x12 | Vertical Display Enable End |
| 0x13 | Offset |
| 0x14 | Underline Location |
| 0x15 | Start Vertical Blanking |
| 0x16 | End Vertical Blanking |
| 0x17 | CRT Mode Control |
| 0x18 | Line Compare |

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:** Cursor Location High Byte

**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 \* 画面幅の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 mov out

al, 0x0f dx, 0x03D4 dx, al

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。

|  |  |  |
| --- | --- | --- |
| mov | al, bl | ; al contains the low byte address |
| mov | dx, 0x03D5 |  |
| out | dx, al | ; low byte |

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。

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;

; MoveCur ()

; - Update hardware cursor

; parm/ bh = Y pos

; parm/ bl = x pos

bits 32

MovCur:

pusha

; save registers (aren't you getting tired of this comment?)

;-------------------------------;

; Get current position ;

;-------------------------------;

; Here, \_CurX and \_CurY are relitave to the current position on screen, not in memory.

; That is, we don't need to worry about the byte alignment we do when displaying characters,

; so just follow the forumla: location = \_CurX + \_CurY \* COLS

;--------------------------------------;

; Set low byte index to VGA register ;

;--------------------------------------;

;---------------------------------------;

; Set high byte index to VGA register ;

;---------------------------------------;

|  |  |  |
| --- | --- | --- |
| xor | eax, eax |  |
| mov | ecx, COLS |  |
| mov | al, bh | ; get y pos |
| mul | ecx | ; multiply y\*COLS |
| add | al, bl | ; Now add x |
| mov | ebx, eax |  |

|  |  |  |
| --- | --- | --- |
| mov | al, 0x0f | ; Cursor location low byte index |
| mov out | dx, 0x03D4 dx, al | ; Write it to the CRT index register |
| mov | al, bl | ; The current location is in EBX. BL contains the low byte, BH high byte |
| mov | dx, 0x03D5 | ; Write it to the data register |
| out | dx, al | ; low byte |

|  |  |  |
| --- | --- | --- |
| xor | eax, eax |  |
| mov | al, 0x0e | ; Cursor location high byte index |
| mov out | dx, 0x03D4 dx, al | ; Write to the CRT index register |
| mov | al, bh | ; the current location is in EBX. BL contains low byte, BH high byte |
| mov | dx, 0x03D5 | ; Write it to the data register |
| out | dx, al | ; high byte |
| popa ret |  |  |

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。

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;

; ClrScr32 ()

; - Clears screen

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*; bits 32

ClrScr32:

pusha cld

mov edi, VIDMEM

mov cx, 2000

mov ah, CHAR\_ATTRIB

mov al, ' '

rep stosw

mov byte [\_CurX], 0

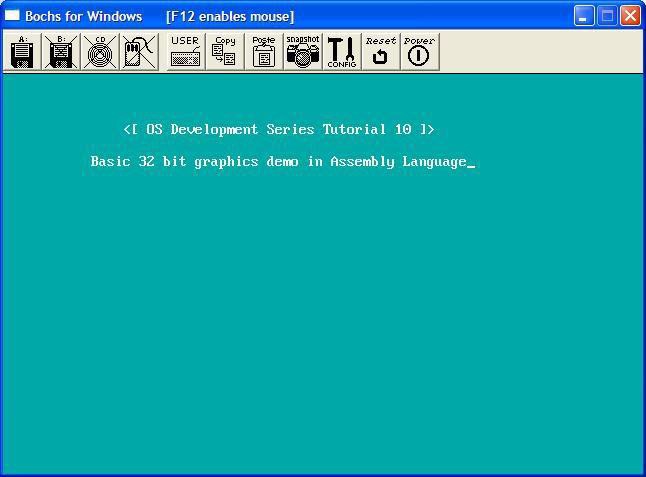
mov byte [\_CurY], 0 popa

ret

簡単でしょう？

これで、テキストを印刷し、ハードウェアカーソルを更新し、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のローダを拡張して、カーネルに制御を与える際に小さなメニュ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。

**Demo**



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しました。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。かっこいいでしょう？

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。かっこ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、基本的なピュアバイナリの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。ご心配なく。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N eptune Operating Systemを開発中 質問やコメントは？ お気軽にご連絡ください。*

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 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 Development Series Tutorial 5: Bootloaders 3](http://www.brokenthorn.com/Resources/OSDev5.html)

[OS Development Series Tutorial 6: Bootloaders 4](http://www.brokenthorn.com/Resources/OSDev6.html)

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ないはずです。

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。

; We are still pure binary. We will fix this in the next few tutorials :)

;---------------------------------------;

; Clear screen and print success ;

;---------------------------------------;

call mov call

ClrScr32 ebx, msg Puts32

;---------------------------------------;

; Stop execution ;

;---------------------------------------;

cli hlt

|  |  |  |  |
| --- | --- | --- | --- |
| org | 0x10000 | ; Kernel starts at 1 MB | |
| bits | 32 | ; 32 bit code | |
| jmp | Stage3 | ; jump to stage 3 | |
| %include "stdio.inc" ; Our stdio.inc file we developed from the previous tutorial msg db 0x0A, 0x0A, "Welcome to Kernel Land!!", 0x0A, 0  Stage3:  ; ;  ; Set registers ;  ; ; | | | |
|  | mov mov mov mov  mov | ax, 0x10 ds, ax ss, ax es, ax  esp, 90000h | ; set data segments to data selector (0x10)  ; stack begins from 90000h |

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る。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、何が起こっ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 = Number of sectors to read

CH = Low eight bits of cylinder number

CL = Sector Number (Bits 0-5). Bits 6-7 are for hard disks only

DH = Head Number

DL = Drive Number (Bit 7 set for hard disks)

ES:BX = Buffer to read sectors to

Returns:

AH = Status Code

AL = Number of sectors read

CF = set if failure, cleared is successfull

これはそれほど難しいことではありません。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いように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。

Okay . 18 sectors per track. Remember that each sector if 512 bytes. Also, remember that there are 80 tracks per side.

Okay then! All of this information. Sectors per track, the number of tracks, number of heads, the size of a sector, completely

depend on the disk itself. Remember that a sector does not NEED to be 512 bytes?

We describe everything in the OEM Parameter Block:

|  |  |  |  |
| --- | --- | --- | --- |
| bpbOEM bpbBytesPerSector: | db DW | "My OS 512 | " |
| bpbSectorsPerCluster: | DB | 1 |  |
| bpbReservedSectors: | DW | 1 |  |
| bpbNumberOfFATs: | DB | 2 |  |
| bpbRootEntries: | DW | 224 |  |
| bpbTotalSectors: | DW | 2880 |  |
| bpbMedia: DB 0xf0 ;; 0xF1  bpbSectorsPerFAT: DW 9  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 " | | | |

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

はセクタに対応しています。しかし、シリンダー（シリンダーはただのヘッドであることを覚えていますか）やトラックでも動作します。

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。

つまり、ロードするセクターを指定したい場合は、リニアセクター番号をディスク上の正確なシリンダー、トラック、セクターの位置に変換する必要が あるのです。

CHSからLBAへの会話の手順を覚えていますか？

LBAからCHSへの変換

これは聞き覚えがあるのではないでしょうか？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。

とにかく、このセクタ番号(LBA)を、ディスク上の正確なシリンダ/ヘッド/セクタの位置に変換する方法を見つけなければなりません。ブートローダ4のチュートリアルで覚えたことがありますか？

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、forumlasをもう一度見てみましょう。

|  |  |  |
| --- | --- | --- |
| absolute sector | = | (logical sector / sectors per track) + 1 |
| absolute head | = | (logical sector / sectors per track) MOD number of heads |
| absolute track | = | logical sector / (sectors per track \* number of heads) |

よしっ! これはかなり簡単だね。論理セクタ」とは、私たちが望む実際のセクタ番号のことです。論理セクタ／トラックあたりのセクタ数は、上記のすべての式の中に入っていることに注意してください。

この除算はすべての式の中にあるので、その結果を保存して、他の2つの式に使うことができます。

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。



absolute sector

=

(logical sector / sectors per track) + 1

2.1111111111111111111111111111111

=

20 / 18 (sectors per track) + 1

絶対数（2）だけを残します。LBAアドレッシングは0から始まるので、ここで1を加える必要があることに注意してください。これらの公式のすべ てに、「論理セクタ/トラックあたりのセクタ」という基本的な公式があることを覚えておいてください。この例では、単純に1.11111111111111111111111111111111（上の式では1を追加していることに注意）です。私たちは整数を扱っているので、これは単に1.



absolute head

=

(logical sector / sectors per track) MOD number of heads

(1) MOD Number of heads (2)

= Head 1

OEMブロックでは、1つのシリンダーに2つのヘッドを指定していました。今のところ、ヘッド1のセクター2を示しています。いいですね。でも、ここはどのコースなんでしょう？



absolute track =

logical sector / (sectors per track \* number of heads)

(1) \* Number of heads (2)

= Track 2

これは、上記とまったく同じ式であることに注意してください。唯一の違いは、この簡単な操作だけです。

とにかく...式に従うと、次のようになります。論理的なセクター20はセクター2トラック2ヘッド0にあります。これを前のセクションで最初に言ったことと比較して、このフォーラムラがどのように機能するかを見てみましょう。）

それでは、これらの数式をコードに適用してみましょう。LBACHS説明。詳細

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（論理セクター

/ 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、単に計算する方が効率的です。

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。



LBACHS:

xor div

dx, dx

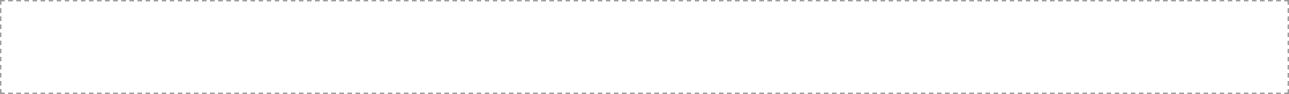
WORD [bpbSectorsPerTrack]

; prepare dx:ax for operation

; calculate

ここでAXには、トラック操作ごとの論理セクタ/セクタが含まれます。

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 mov

dl

BYTE [absoluteSector], dl

; adjust for sector 0

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

; prepare dx:ax for operation

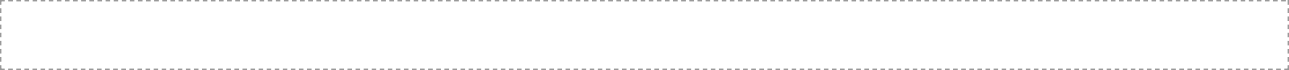
さて、数式ですが...。

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 アブソリュート・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]

; calculate

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には同じ操作のREMAINDER(Modolus)が格納されます。）



mov mov ret

BYTE [absoluteHead], dl BYTE [absoluteTrack], al

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。



ClusterLBA:

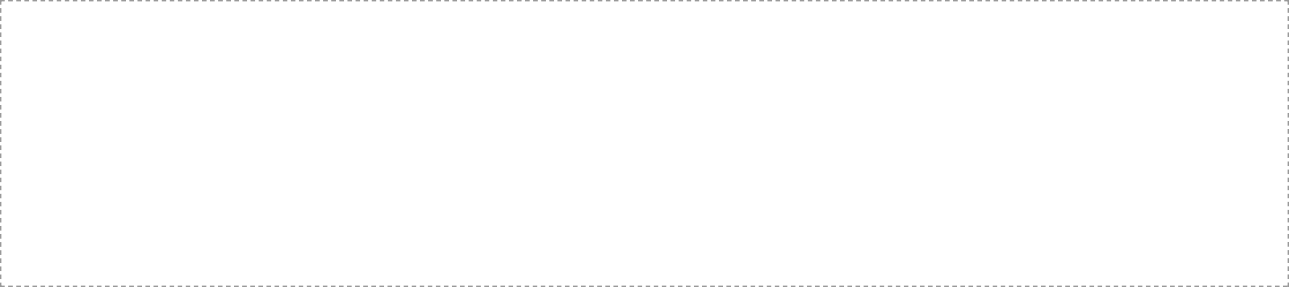
; LBA =

(cluster - 2 ) \* sectors per cluster

|  |  |  |
| --- | --- | --- |
| sub | ax, 0x0002 | ; subtract 2 from cluster number |
| xor | cx, cx |  |
| mov | cl, BYTE [bpbSectorsPerCluster] | ; get sectors per cluster |
| mul | cx | ; multply |

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。



;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;

; Reads a series of sectors

; CX=>Number of sectors to read

; AX=>Starting sector

; ES:BX=>Buffer to read to

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;

ReadSectors:

.MAIN

mov

di, 0x0005

; five retries for error

さて、ここではセクタを5回読むことを試みます。



.SECTORLOOP

|  |  |  |
| --- | --- | --- |
| push | ax |  |
| push | bx |  |
| push | cx |  |
| call | LBACHS | ; convert starting sector to CHS |

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？

|  |  |  |
| --- | --- | --- |
| mov | ah, 0x02 | ; BIOS read sector |
| mov | al, 0x01 | ; read one sector |
| mov | ch, BYTE [absoluteTrack] | ; track |
| mov | cl, BYTE [absoluteSector] | ; sector |
| mov | dh, BYTE [absoluteHead] | ; head |
| mov | dl, BYTE [bsDriveNumber] | ; drive |
| int | 0x13 | ; invoke BIOS |

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

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  xor | .SUCCESS  ax, ax | ; test for read error  ; BIOS reset disk |
| int | 0x13 | ; invoke BIOS |
| dec | di | ; decrement error counter |
| pop | cx |  |
| pop | bx |  |
| pop | ax |  |
| jnz | .SECTORLOOP | ; attempt to read again |

BIOSのINT0x13関数2は、エラーがあるとキャリーフラグ(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 |  |  |
| pop | cx |  |
| pop | bx |  |
| pop | ax |  |
| add | bx, WORD [bpbBytesPerSector] | ; queue next buffer |
| inc | ax | ; queue next sector |
| loop | .MAIN | ; read next sector |
| ret |  |  |

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。難しいことではありません) 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で、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。うわぁ... :( 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。このような理由から、このチュートリアルではFAT12のコードについて説明することにしました。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ました。このため、ここでは詳細に説明することにしました。

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

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Boot Sector | Extra Reserved Sectors | File Allocation Table 1 | File Allocation Table 2 | Root Directory (FAT12/FAT16 Only) | Data Region containng files and directories. |

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、どこに何があるかを物理的に追跡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メモリマネージャです。詳しくは後ほど...

es

; store registers

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; LoadRoot ()

; - Load Root Directory Table

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* LoadRoot:

pusha push

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で、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。単純なことですが、セクター数が必要なので、この結果をセクター数で割る必要があります。

; compute size of root directory and store in "cx"

|  |  |  |
| --- | --- | --- |
| xor | cx, cx | ; clear registers |
| xor | dx, dx |  |
| mov | ax, 32 | ; 32 byte directory entry |
| mul | WORD [bpbRootEntries] | ; total size of directory |
| div | WORD [bpbBytesPerSector] | ; sectors used by directory |
| xchg | ax, cx | ; move into AX |

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。

|  |  |  |
| --- | --- | --- |
| mov | al, byte [bpbNumberOfFATs] | ; number of FATs |
| mul | word [bpbSectorsPerFAT] | ; sectors used by FATs |

|  |  |  |  |
| --- | --- | --- | --- |
| add | ax, word [bpbReservedSectors] | ; adjust for bootsector | |
| mov | word [datasector], ax | ; base of | root directory |
| add | word [datasector], cx |  | |

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。

; read root directory

|  |  |  |
| --- | --- | --- |
| push | word ROOT\_SEG |  |
| pop | es |  |
| mov | bx, 0x0 | ; copy root dir |
| call | ReadSectors | ; read in directory table |
| pop | es |  |
| popa |  | ; restore registers and return |
| ret |  |  |

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。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Boot Sector | Extra Reserved Sectors | File Allocation Table 1 | File Allocation Table 2 | Root Directory (FAT12/FAT16 Only) | Data Region containng files and directories. |

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。

es

; store registers

pusha push

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

LoadFAT:

Parm/ ES:DI => Root Directory Table

- Loads FAT table

;

;

;

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; LoadFAT ()

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。

; compute size of FAT and store in "cx"

|  |  |  |
| --- | --- | --- |
| xor | ax, ax |  |
| mov | al, BYTE [bpbNumberOfFATs] | ; number of FATs |
| mul | word [bpbSectorsPerFAT] | ; sectors used by FATs |
| mov | cx, ax |  |

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...。

ax, word [bpbReservedSectors]

mov

; compute location of FAT and store in "ax"

よっしゃー! 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を呼び出してください。

; restore registers and return

es

pop popa ret

; read FAT into memory (Overwrite our bootloader at 0x7c00)

push word FAT\_SEG

pop es

xor bx, bx call ReadSectors

それが全てです。）

ファイルの検索

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。

**Bytes 0-7 : DOS File name (Padded with spaces) Bytes 8-10 : DOS File extension (Padded with spaces) Bytes 11 :** File attributes. This is a bit pattern:

**Bit 0 :** Read Only

**Bit 1 :** Hidden

**Bit 2 :** System

**Bit 3 :** Volume Label

**Bit 4 :** This is a subdirectory

**Bit 5 :** Archive

**Bit 6 :** Device (Internal use)

**Bit 6 :** Unused

**Bytes 12 :** Unused

**Bytes 13 :** Create time in ms

**Bytes 14-15 :** Created time, using the following format:

**Bit 0-4 :** Seconds (0-29)

**Bit 5-10 :** Minutes (0-59)

**Bit 11-15 :** Hours (0-23)

**Bytes 16-17 :** Created year in the following format:

**Bit 0-4 :** Year (0=1980; 127=2107

**Bit 5-8 :** Month (1=January; 12=December)

**Bit 9-15 :** Hours (0-23)

**Bytes 18-19 :** Last access date (Uses same format as above)

**Bytes 20-21 :** EA Index (Used in OS/2 and NT, dont worry about it)

**Bytes 22-23 :** Last Modified time (See byte 14-15 for format)

**Bytes 24-25 :** Last modified date (See bytes 16-17 for format)

**Bytes 26-27 : First Cluster**

**Bytes 28-32 : File Size**

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。

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; FindFile ()

; - Search for filename in root table

;

; parm/ DS:SI => File name

; ret/ AX => File index number in directory table. -1 if error

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* FindFile:

|  |  |  |
| --- | --- | --- |
| push | cx | ; store registers |
| push | dx |  |
| push | bx |  |
| mov | bx, si | ; copy filename for later |

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？

そのためには、エントリーの数を知る必要があります。

; browse root directory for binary image

|  |  |  |
| --- | --- | --- |
| mov | cx, word [bpbRootEntries] | ; load loop counter |
| mov | di, ROOT\_OFFSET | ; locate first root entry |
| cld |  | ; clear direction flag |

さて、これでCXには調べるべきエントリの数が入りました。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で、これを解析してみましょう。

|  |  |  |
| --- | --- | --- |
| .LOOP: |  |  |
| push | cx |  |
| mov | cx, 11 | ; eleven character name. Image name is in SI |
| mov | si, bx | ; image name is in BX |

push di rep cmpsb

; test for entry match

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？)

; queue next directory entry

di

.Found cx

di, 32

.LOOP

pop je pop add loop

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:

|  |  |  |
| --- | --- | --- |
| pop | bx | ; restore registers and return |
| pop | dx |  |
| pop | cx |  |
| mov | ax, -1 | ; set error code |
| ret |  |  |

ファイルが見つかった場合は、すべてのレジスターを復元します。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。

|  |  |  |
| --- | --- | --- |
| .Found: |  |  |
| pop | ax | ; return value into AX contains entry of file |
| pop | bx | ; restore registers and return |
| pop | dx |  |
| pop | cx |  |
| ret |  |  |

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

これでようやくすべての設定が完了し、いよいよファイルの読み込みです。

この部分のほとんどは、他のルーチンを呼び出しているので、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。

ecx, ecx ecx

xor push

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; LoadFile ()

; - Load file

; parm/ ES:SI => File to load

; parm/ BX:BP => Buffer to load file to

; ret/ AX => -1 on error, 0 on success

; ret/ CX => Number of sectors loaded

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

LoadFile:

ここではレジスター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（当たり前ですよね＾＾）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。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| push push | | bx ; B  bp | X=>BP points to buffer to write to; store it for later | |
| call | | FindFile | ; find our file. ES:SI contains our filename | |
| cmp | | ax, -1 | ; check for error | |
| jne | | .LOAD\_IMAGE | \_PRE ; No error :) Load the FAT | |
| pop pop  pop | bp bx  ecx | | | ; Nope :( Restore registers, set error code and return | |

mov ret

ax, -1

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...。

; store location for later again

; Refrence the table for file's first cluster

; get location to write to so we dont screw up the stack

word [cluster], dx bx

es bx es

word ROOT\_SEG es

dx, word [es:di + 0x001A]; ES:DI points to file entry in root directory table.

push pop mov mov pop pop push push

.LOAD\_IMAGE\_PRE:

sub edi, ROOT\_OFFSET

sub eax, ROOT\_OFFSET

; get starting cluster

上記は面倒くさいですよね。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...。

call LoadFAT

; Load the FAT to 0x7c00

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:

mov pop pop call xor mov call

ax, WORD [cluster] es

bx ClusterLBA cx, cx

; cluster to read

; convert cluster to LBA

cl, BYTE [bpbSectorsPerCluster] ; sectors to read

ReadSectors

; Read in cluster

pop inc push

ecx ecx ecx

; increment sector count

push push

bx es

; save registers for next iteration

mov mov xor

ax, FAT\_SEG es, ax

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、トラックとヘッドの情報を持っていることを前提としています。ReadSectors()ではLBAのリニアセクター番号が必要なので、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、クラスター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。

; compute next cluster

|  |  |  |
| --- | --- | --- |
| mov | ax, WORD [cluster] | ; identify current cluster |
| mov | cx, ax | ; copy current cluster |

|  |  |  |
| --- | --- | --- |
| mov | dx, ax | ; copy current cluster |
| shr | dx, 0x0001 | ; divide by two |

|  |  |  |  |
| --- | --- | --- | --- |
| add | cx, dx | ; | sum for (3/2) |
| mov | bx, 0 | ; | location of FAT in memory |
| add | bx, cx | ; | index into FAT |
| mov  test jnz | dx, WORD [es:bx] ax, 0x0001  .ODD\_CLUSTER | ; | read two bytes from FAT |

|  |  |  |  |
| --- | --- | --- | --- |
| and jmp | dx, 0000111111111111b  .DONE | ; | take low twelve bits |
| .ODD\_CLUSTER: | | | |
| shr | dx, 0x0004 | ; | take high twelve bits |
| .DONE:  mov | WORD [cluster], dx | ; | store new cluster |
| cmp | dx, 0x0FF0 | ; | test for end of file marker (0xFF) |
| jb | LOAD\_IMAGE | ; | No? Go on to next cluster then |
| DONE:  pop | es | ; | restore all registers |
| pop pop xor  ret | bx ecx  ax, ax | ; | return success code |

これが全てです。少し複雑ですが、難しいことではないと思います。）

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。

|  |  |  |  |
| --- | --- | --- | --- |
| call  mov | LoadRoot  ebx, 0 | ;  ; | Load root directory table  BX:BP points to buffer to load to |
| mov | ebp, IMAGE\_RMODE\_BASE |  |  |
| mov | Esi, ImageName | ; | our file to load |
| call | LoadFile | ; | load our file |
| MOV | dword [ImageSize], ecx | ; | size of kernel |
| cmp | ax, 0 | ; | Test for success |
| je | EnterStage3 | ; | yep--onto Stage 3! |
| mov | si, msgFailure | ; | Nope--print error |
| call | Puts16 |  |  |
| mov | ah, 0 |  |  |
| int | 0x16 | ; | await keypress |
| int | 0x19 | ; | warm boot computer |
| cli |  | ; | If we get here, something really went wong |
| hlt |  |  |  |

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。

|  |  |  |
| --- | --- | --- |
| bits 32 |  |  |
| Stage3: |  |  |
| mov mov mov mov mov | ax, DATA\_DESC  ds, ax ss, ax es, ax  esp, 90000h | ; set data segments to data selector (0x10)  ; stack begins from 90000h |

; Copy kernel to 1MB (0x10000)

CopyImage:

mov eax, dword [ImageSize]

movzx ebx, word [bpbBytesPerSector]

mul mov div cld mov mov mov rep

ebx ebx, 4 ebx

esi, IMAGE\_RMODE\_BASE edi, IMAGE\_PMODE\_BASE

ecx, eax movsd

; copy image to its protected mode address

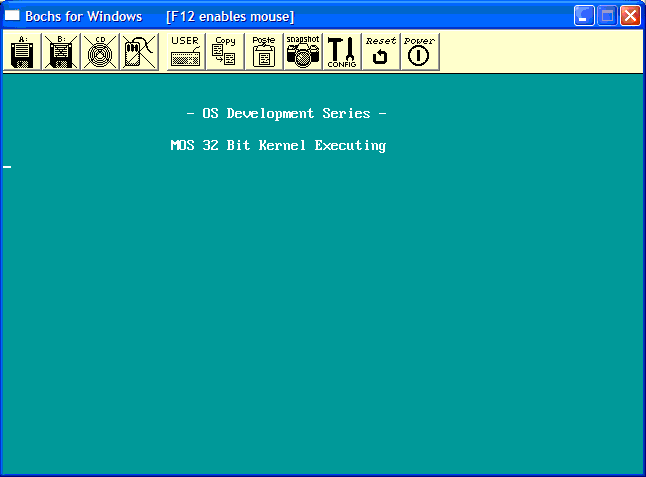
call CODE\_DESC:IMAGE\_PMODE\_BASE; execute our kernel!

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？

**Demo**



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。すごいでしょう？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」というものもありません。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。

*See you there ;)*

次の機会まで。

~マイク

*BrokenThorn Entertainment社。現在、DoEとN eptune Operating Systemを開発中 質問やコメントは？お気軽にご連絡ください。*

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。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。辞書では、「カーネル」を「核」、「本質的な部分」、あるいは「何かの本体」と定義しています。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か 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した。

では、カーネルは何のために存在するのでしょうか。計算機環境では、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。そうなると、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などというものは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レイヤーを提供します。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でしょうか？そうなると、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はなく、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し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）

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か？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、メモリー、プロセッサー（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ーや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

デバイスドライバーは、カーネルを経由してハードウェアにアクセスします。デバイスドライバーは、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、ドライバーが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、これらのドライバーが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イ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考えてみましょう。太字のものはすべてカーネルが担当しています。

**Memory Management Program Management Multitasking**

**Memory Protection**

Fixed Base Address - This was covered in Tutorial 2

Multiuser - This is useually implimented by a shell

**Kernel** - Of course **File System** Command Shell

Graphical User Interface (GUI) Graphical Shell

Linear Block Addressing (LBA) - This was covered in Tutorial 2

Bootloader -Completed

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ーとしてntfs.sysを使用しています。

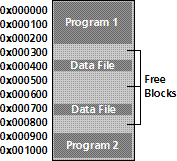
このリストは、チュートリアル2で見覚えがあるはずです。また、これらの用語のいくつかを取り上げました。太字の用語を見て、それらがどのようにカーネルに関連しているかを見てみましょう。また、いくつかの新しい概念についても見ていきます。

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。



ここで何が起こっ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 Indipendent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

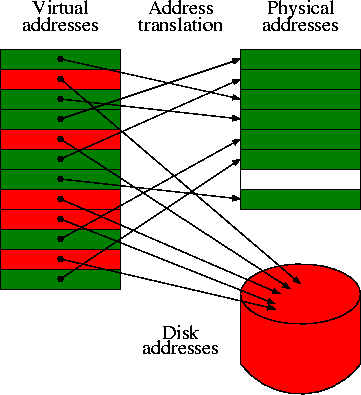
に扱うことができます。

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する方法でもあります。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。



ここで何が起こっ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。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.0 :)

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。

プログラムマネジメント

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。

プロセッサー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いますか？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ったことも覚えています。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はスーパバイザ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。）*

スーパバイザモードでは、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。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、ファーコール、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るのです。

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。

Bit 0 (Bit 40 in GDT): Access bit (Used with Virtual Memory). Because we don't use virtual memory (Yet, anyway), we will ignore it. Hence, it is 0

Bit 1 (Bit 41 in GDT): is the readable/writable bit. Its set (for code selector), so we can read and execute data in the segment (From 0x0 through 0xFFFF) as code

Bit 2 (Bit 42 in GDT): is the "expansion direction" bit. We will look more at this later. For now, ignore it.

Bit 3 (Bit 43 in GDT): tells the processor this is a code or data descriptor. (It is set, so we have a code descriptor)

Bit 4 (Bit 44 in GDT): Represents this as a "system" or "code/data" descriptor. This is a code selector, so the bit is set to 1.

Bits 5-6 (Bits 45-46 in GDT): is the privilege level (i.e., Ring 0 or Ring 3). We are in ring 0, so both bits are 0.

Bit 7 (Bit 47 in GDT): Used to indicate the segment is in memory (Used with virtual memory). Set to zero for now, since we are not using virtual memory yet

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Global Descriptor Table (GDT)

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* gdt\_data:

; Null descriptor (Offset: 0x0)--Remember each descriptor is 8 bytes! dd 0 ; null descriptor

dd 0

; Kernel Space code (Offset: 0x8 bytes)

dw 0FFFFh ; limit low

dw 0 ; base low

db 0 ; base middle

db 10011010b ; access - Notice that bits 5 and 6 (privilege level) are 0 for Ring 0 db 11001111b ; granularity

db 0 ; base high

; Kernel Space data (Offset: 16 (0x10) bytes

dw 0FFFFh ; limit low (Same as code)10:56 AM 7/8/2007

dw 0 ; base low

db 0 ; base middle

db 10010010b ; access - Notice that bits 5 and 6 (privilege level) are 0 for Ring 0 db 11001111b ; granularity

db 0 ; base high

; User Space code (Offset: 24 (0x18) bytes)

dw 0FFFFh ; limit low

dw 0 ; base low

db 0 ; base middle

db 11111010b ; access - Notice that bits 5 and 6 (privilege level) are 11b for Ring 3 db 11001111b ; granularity

db 0 ; base high

; User Space data (Offset: 32 (0x20) bytes

dw 0FFFFh ; limit low (Same as code)10:56 AM 7/8/2007

dw 0 ; base low

db 0 ; base middle

db 11110010b ; access - Notice that bits 5 and 6 (privilege level) are 11b for Ring 3

db 11001111b

db 0

; granularity

; base high

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、ファーコール、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。

; Call gate (Offset: 24 (0x18) bytes

CallGate1:

dw (Gate1 & 0xFFFF) ; limit low address of gate routine dw 0x8 ; code segment selector

db 0 ; base middle

db 11101100b ; access - Notice that bits 5 and 6 (privilege level) are 11 for Ring 3 db 0 ; granularity

db (Gate1 >> 16) ; base high of gate routine

; End of the GDT. Define the routine wherever

; The call gate routine Gate1:

; do something special here at Ring 3

retf ; far return back to calling routine

; limit low (Same as code)10:56 AM 7/8/2007

; base low

; base middle

; access - Notice that bits 5 and 6 (privilege level) are 0 for Ring 0

; granularity

; base high

; Kernel Space data (Offset: 16 (0x10) bytes dw 0FFFFh

dw 0

db 0

db 10010010b db 11001111b

db 0

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

; Global Descriptor Table (GDT)

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* gdt\_data:

; Null descriptor (Offset: 0x0)--Remember each descriptor is 8 bytes! dd 0 ; null descriptor

dd 0

; Kernel Space code (Offset: 0x8 bytes)

dw 0FFFFh ; limit low

dw 0 ; base low

db 0 ; base middle

db 10011010b ; access - Notice that bits 5 and 6 (privilege level) are 0 for Ring 0 db 11001111b ; granularity

db 0 ; base high

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。

; far call--calls our Gate1 routine

; execute the call gate

call far 0x18:0

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 + the value 8

ESP = IA32\_SYSENTER\_ESP MSR EIP = 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 + the value 16

ESP = ECX Register

EIP = EDX Register

SS = IA32\_SYSENTER\_CS MSR MSR + 24

SYSENTER/SYSEXITの使い方

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。

mov eax, esp

mov edx, 0

mov ecx, IA32\_SYSENTER\_ESP wrmsr

mov eax, Sysenter\_Entry mov edx, 0

mov ecx, IA32\_SYSENTER\_EIP wrmsr

; Now, we can use sysenter to execute Sysenter\_Entry at ring 0 from either a Ring 0 program or Ring 3: sysenter

Sysenter\_Entry:

; sysenter jumps here, is is executing this code at prividege level 0. Simular to Call Gates, normally we will

; provide a single entry point for all system calls.

; kernel code descriptor

eax, 0x8 edx, 0

ecx, IA32\_SYSENTER\_CS

mov mov mov wrmsr

%define IA32\_SYSENTER\_CS 0x174

%define IA32\_SYSENTER\_ESP 0x175

%define IA32\_SYSENTER\_EIP 0x176

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るのでしょうか？

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で、元々問題を起こしたプログラムを探してみることができます。

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、カーネルの理論、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なぞっただけです。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イ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N eptune Operating Systemを開発中 質問やコメントは？お気軽にご連絡ください。*

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。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では、先にチュートリアル2を読んでおくことをお勧めします。

*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思っています。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りますか？T utorial 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

よし、それならば! 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基本的な項目のリストを作成しました。それをもう一度、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、T utorial 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ると思います。

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。

プロセッサーマネージメント

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。このインターフェースは、通常、デバイスドライバーやハードウェア 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。

ここで登場するのが、「プログラム・マネジメント」と「マルチタスク」です。これらの用語はチュートリアル2 でお馴染みですね。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、そしてチュートリアル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きたはずです。しかし、まだ説明していないことがあります。

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

いやぁ、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 :)

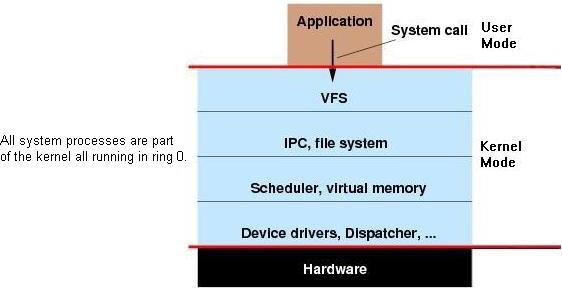
カーネルデザイン - 概要：一次設計モデル

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（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してしまいます。

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。Unixライクなカーネル

Linux シラブル Unix カーネル

DOS

BSD

Solaris AI

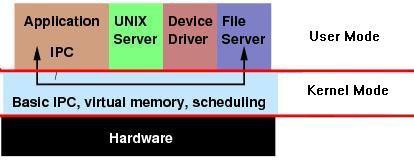
DR-DOS MS-DOS

FreeBSD NetBSD

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 ー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ー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、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で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。かっこいいですよね。

*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。

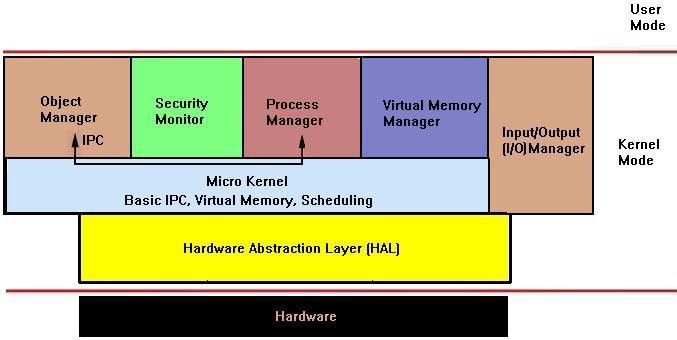
非同期IPC

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いてください。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

[ww](http://www.brokenthorn.com/Resources/OSDev13.html) 11

[w.brokenthorn.com/Resources/OSDev13.html](http://www.brokenthorn.com/Resources/OSDev13.html)

*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 です。

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、いくつかの論争があります。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 を作ることもできます。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N eptune Operating Systemを開発中。*

*ご質問やご意見がございましたら お気軽にお問い合わせください。*

記事をより良くするために、あなたも貢献してみませんか？よろしければ、ぜひご連絡ください。



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 - 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）。

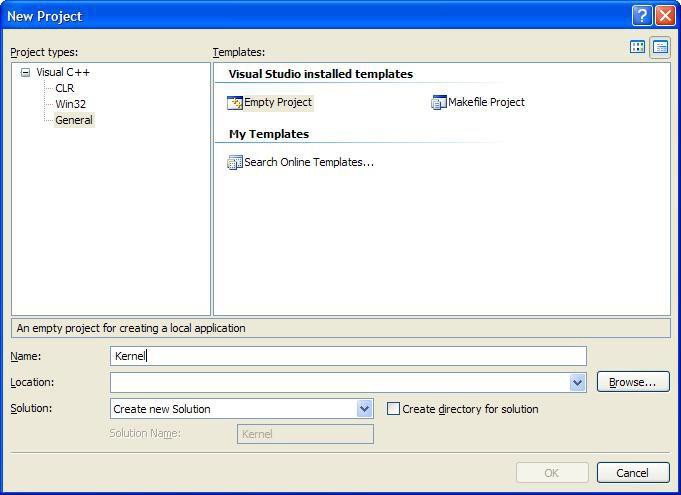
ただし、C++の機能の中には、私たちがまだ実装していない詳細情報に依存しているものもありますのでご注意ください。これらの詳細はかなり高度なものです。例えば、C++ の

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。



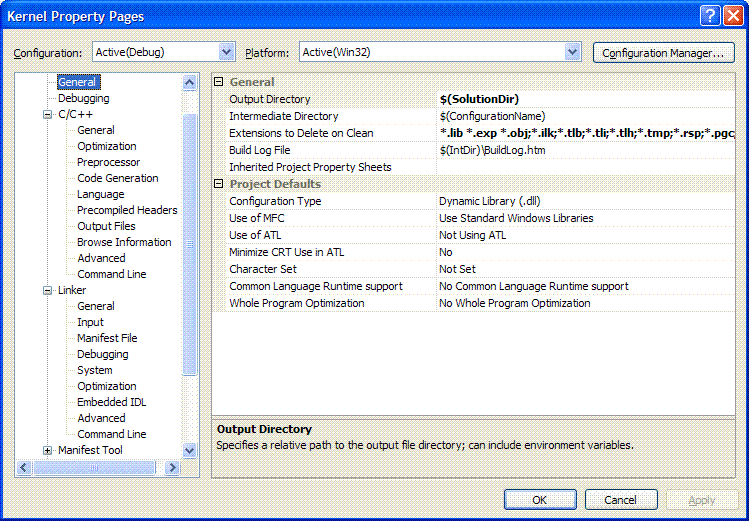
インストールされたテンプレート」で「空のプロジェクト」の設定が選択されていることに注意してください。

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。

#### 構築環境 - プロジェクトのプロパティ

これは素晴らしいですね。しかし問題は、このコンパイラー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るはずです。

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てしまいます。

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（「Configuation 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・ダイレクトリー

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>）を区別する際に役立ちます。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。デバッガは、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、今はデバッガ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よね？）

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、これを参考にしてみてください。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。

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って非常に重要です。リンカは、コンパイラー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。

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（または仮想の）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

標準ライブラリがないので、Yes（/NODEFAULTLIB）に設定してください。

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。本製品はドライバーアプリケーションなので、この値をネイティブ（/SUBSYSTEM:NATIVE）です。

ドライバー

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。ここではドライバー・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 redundent 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。私たちのシステムでは、kernel\_entryに なります。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。

/OUT:"A:\KRNL32.EXE" /INCREMENTAL:NO /NOLOGO /LIBPATH:"..\Lib\\" /MANIFEST:NO

/NODEFAULTLIB /MAP:"Kernel.map" /SUBSYSTEM:NATIVE /DRIVER /OPT:REF /OPT:ICF /ENTRY:"kernel\_entry"

/BASE:"0x100000" /FIXED:No /ERRORREPORT:PROMPT

#### PE カーネルの実行

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。プログラムマネージャやタスクマネージャの話をするまではね。

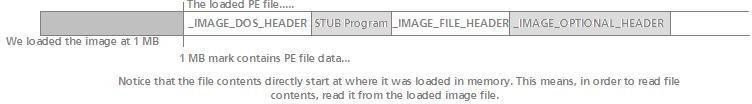
しかし問題は、MSVC++はCOFFとPEのファイルフォーマットしか出力できないことです。このため、Stage 2ブートローダの中で解析する方法を見つけなければなりません。

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 header USHORT e\_magic; // Magic number (Should be MZ) USHORT e\_cblp; // Bytes on last page of file USHORT e\_cp; // Pages in file

USHORT e\_crlc; // Relocations

USHORT e\_cparhdr; // Size of header in paragraphs

|  |  |  |  |
| --- | --- | --- | --- |
| USHORT  USHORT | e\_minalloc;  e\_maxalloc; | //  // | Minimum extra paragraphs needed  Maximum extra paragraphs needed |
| USHORT | e\_ss; | // | Initial (relative) SS value |
| USHORT | e\_sp; | // | Initial SP value |
| USHORT | e\_csum; | // | Checksum |
| USHORT | e\_ip; | // | Initial IP value |
| USHORT | e\_cs; | // | Initial (relative) CS value |
| USHORT | e\_lfarlc; | // | File address of relocation table |
| USHORT | e\_ovno; | // | Overlay number |
| USHORT | e\_res[4]; | // | Reserved words |
| USHORT | e\_oemid; | // | OEM identifier (for e\_oeminfo) |
| USHORT | e\_oeminfo; | // | OEM information; e\_oemid specific |
| USHORT | e\_res2[10]; | // | Reserved words |
| LONG | e\_lfanew; | // | File address of new exe 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、エントリー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ー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 is a 4 byte offset address of the PE header; it is 60th byte. Get it |
| add | ebx, IMAGE\_PMODE\_BASE | ; add base |

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と、「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した。

\_image\_file\_header

typedef struct \_IMAGE\_FILE\_HEADER

{ USHORT Machine;

USHORT NumberOfSections;

ULONG TimeDateStamp;

ULONG PointerToSymbolTable; ULONG NumberOfSymbols; USHORT SizeOfOptionalHeader; USHORT Characteristics;

} IMAGE\_FILE\_HEADER, \*PIMAGE\_FILE\_HEADER;

さて...EBXにはこの構造体の開始アドレスが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で、今 はこの構造体をスキップすることができます。

; EBX now points to beginning of \_IMAGE\_FILE\_HEADER. Jump over it to the next section (\_IMAGE\_OPTIONAL\_HEADER) add ebx, 24

; e\_lfanew is a 4 byte offset address of the PE header; it is 60th byte. Get it

; add base

ebx, [IMAGE\_PMODE\_BASE+60] ebx, IMAGE\_PMODE\_BASE

mov add

\_image\_optional\_header

// NT additional fields.

//

ULONG ImageBase;

<< IMPORTANT!

MajorLinkerVersion; MinorLinkerVersion; SizeOfCode; SizeOfInitializedData; SizeOfUninitializedData; AddressOfEntryPoint; BaseOfCode;

BaseOfData;

UCHAR UCHAR ULONG ULONG ULONG ULONG ULONG ULONG

//

struct \_IMAGE\_OPTIONAL\_HEADER {

//

// Standard fields.

//

USHORT Magic;

ULONG SectionAlignment; ULONG FileAlignment;

USHORT MajorOperatingSystemVersion; USHORT MinorOperatingSystemVersion; USHORT MajorImageVersion;

USHORT MinorImageVersion; USHORT MajorSubsystemVersion; USHORT MinorSubsystemVersion; ULONG Reserved1;

ULONG SizeOfImage; ULONG SizeOfHeaders; ULONG CheckSum; USHORT Subsystem;

USHORT DllCharacteristics; ULONG SizeOfStackReserve; ULONG SizeOfStackCommit; ULONG SizeOfHeapReserve; ULONG SizeOfHeapCommit; ULONG LoaderFlags;

ULONG NumberOfRvaAndSizes;

IMAGE\_DATA\_DIRECTORY DataDirectory[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というメンバーで、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を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おいてください。

|  |  |  |
| --- | --- | --- |
| mov | ebx, [IMAGE\_PMODE\_BASE+60] | ; e\_lfanew is 60th byte. Get it |
| add | ebx, IMAGE\_PMODE\_BASE | ; Add base address. EBX now points to file sig (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。

; jump over to optional header (Although it isnt optional o.0 )

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| add | ebx, | 24 |  | | | | | | | | | |
| mov | eax, | [ebx] | ; | \_IMAGE\_FILE\_HEADER is | 20 | bytes | + | size | of | sig | (4 | bytes) |
| add | ebx, | 16 | ; | address of entry point | is | now | in | ebx |  |  |  |  |

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\_OPTONAL\_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メンバーが見えてきます。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ので、これはとても簡単です。

|  |  |  |
| --- | --- | --- |
| mov | ebp, dword [ebx] | ; store entry point address |
| add | ebx, 12 | ; ImageBase member is 12 bytes from AddressOfEntryPoint member |
| mov | eax, dword [ebx] | ; gets image base |
| add | ebp, eax | ; add image base to entry point address |

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 ; Execute Kernel

難しくないでしょう？ここでは、コードセレクタ（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、Windows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 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。

void\* cdecl ::operator new (unsigned int size) { return 0; } void\* cdecl operator new[] (unsigned int size) { return 0; } void cdecl ::operator delete (void \* p) {}

void cdecl operator delete[] (void \* p) { }

これで、新しいキーワードや削除キーワードをエラーなく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。

純粋な仮想関数呼び出しハンドラ

純粋仮想関数とは、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。

[a] ebx, a

fistp mov ret

int a; #ifdef ARCH\_X86

\_asm

\_asm

\_asm

#endif

}

extern "C" long declspec (naked) \_ftol2\_sse() {

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けることができるということです。

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？

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るのです。

.section\_name$location\_name

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す。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。

.CRT$XCA

.CRT$XCU

.CRT$XCZ

各部がアルファベットのよう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。

別の見方をすると、これらを混ぜ合わせてみると...。

.CRT$XCZ

.CRT$XCA

.CRT$XCU

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ことが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。

#pragma data\_seg(".CRT$XCA")

// All varables allocated here are now placed within the .CRT$XCA section, rather then .data section

デフォルト（.data）のセクションに戻すには、このプラグマをパラメータなしで使用します。

//! Select the default data segment again (.data) for the rest of the unit #pragma data\_seg()

セクションの統合

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。

//! Now, move the CRT data into .data section so we can read/write to it #pragma comment(linker, "/merge:.CRT=.data")

なるほど...。では、次のように考えてみましょう。すべてのグローバルな初期化ルーチンは、バイナリイメージの.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。このセクションの前後にセクションを宣言すると、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 っているので、これらのセクションを指す変数を宣言することで、実質的にイニ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。

//! Function pointer typedef for less typing typedef void ( cdecl \*\_PVFV)(void);

まず、読みやすさを向上させるために、関数ポインタを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。

/\*\*

\*

\*

\*

\*/

MSVC++ creates dynamic initializers and deallocaters, which help us in calling the routines. The compilier and linker bind all dynamic initializers into a function pointer table inside a section called .CRT$XCU.

// Standard C++ Runtime (STD CRT) xc\_a points to beginning of initializer table

#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 is located here.

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。

//! Standard C++ Runtime (STD CRT) xc\_z points to end of initializer table #pragma data\_seg(".CRT$XCZ")

\_PVFV xc\_z[] = { NULL };

これは、.CRT$XCZのセクションです。ここも命名規則により、.CRT$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。

//! Select the default data segment again (.data) for the rest of the unit #pragma data\_seg()

その他のデータについては、「.data」セクションを使用したいので、そのセクションに切り替えてください...。

//! Now, move the CRT data into .data section so we can read/write to it #pragma comment(linker, "/merge:.CRT=.data")

...そして、.CRTセクションを.dataセクションに統合します。これにより、.data セクションから .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には、各関数ポインタをループして呼び出すだけです。警告。ヌルの関数ポインター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。

void cdecl \_initterm ( \_PVFV \* pfbegin, \_PVFV \* pfend )

{

// Go through each initializer while ( pfbegin < pfend )

{

// Execute the global initializer if ( \*pfbegin != NULL )

(\*\*pfbegin) ();

// Go to next initializer inside the initializer table

++pfbegin;

}

}

// This initializes all global initializer routines:

\_initterm( xc\_a, xc\_z);

環境の浄化

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。

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。

//! function pointer table to global deinitializer table static \_PVFV \* pf\_atexitlist = 0;

//! Maximum entries allowed in table

static unsigned max\_atexitlist\_entries = 32;

//! Current amount of entries in table static unsigned cur\_atexitlist\_entries = 0;

これらは、グローバルなデイニシャライザー配列のどこにいるのかを追跡するために使用する関数ポインタです。

これらの配列がどこにあるかを定義します。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追加します。これは、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。

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。

実際のルーチンは簡単です。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、必要なことは dtor の配列の最後に追加するだけです。

//! For every global object created, MSVC++ calls this routine with a function ptr to each dtor int cdecl atexit(\_PVFV fn)

{

//! Insure we have enough free space

if (cur\_atexitlist\_entries>=max\_atexitlist\_entries) return 1;

else {

// Add the exit routine

\*(pf\_atexitlist++) = fn; cur\_atexitlist\_entries++;

}

return 0;

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。

void cdecl \_atexit\_init(void)

{

max\_atexitlist\_entries = 32;

// Warning: Normally, the STDC will dynamically allocate this. Because we have no memory manager, just choose

// a base address that you will never use for now pf\_atexitlist = (\_PVFV \*)0x500000;

}

難しいことではないと思います。）

しかし、読者にとっては新しい概念がたくさんあるので、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。

#### エントリーポイント

さて、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セットアップ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 // clear interrupts--Do not enable them yet

mov ax, 10h // offset 0x10 in gdt for data selector, remember? mov ds, ax

mov es, ax mov fs, ax mov gs, ax

mov ss, ax // Set up base stack 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。

mov ebp, esp push ebp

}

#endif

では、main()ルーチンを呼び出しましょう。

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）。

//! Execute global constructors InitializeConstructors();

//! Call kernel entry point main ();

//! Cleanup all dynamic dtors Exit ();

#ifdef ARCH\_X86

\_asm cli

#endif

for (;;);

}

これで必要なものはすべて揃いました。エントリーポイントが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--1MBに設定されてい るはずです

**Demo**

D emダoウンロード（MSVC+）

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のすべてのソースコードも含まれています。

#### 結論

やったー このチュートリアルのコンセプトの多くは、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ー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た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。うまくいったでしょうか？）

次の機会まで。

~マイク

*BrokenThorn Entertainment社。現在、DoEとN eptune Operating Systemを開発中 質問やコメントは？お気軽にご連絡ください。*

記事をより良くするために、あなたも貢献してみませんか？よろしければ、ぜひご連絡ください。



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 cctype

CRT: va\_list.h, stdarg.h/csdtarg

デモ。Debug Printfの書き方（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歩です。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しまうことがあります。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、この醜く書かれた、設計が不十分なコードの混沌とした性質に依存す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、箱を閉じて他のシステムと接続します。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、コンパイルエラー、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。

インターフェースとインプリメンテーション

再び「ポンドラの箱」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。箱のインターフェース（「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

スとをつなぐものです。インターフェース自体には、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ー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するの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。インターフェースを作るには

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。

覚えておいてください。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、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ポ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。

Our two stage bootloader (We have already constructed this from our previous tutorials)

=======================================

SysBoot\

Stage1\

* Stage1 bootstrap loader
* Stage2 KRNLDR bootloder

Our System Core

=======================================

SysCore\

Debug\ - Pre-Release complete builds

Release\ - Release builds

Include\ - Standard Library Include directory

Lib\ Ha l\ Kern el\

* Standard Library Runtime. Outputs Crtlib.lib or Crtlib.dll.
* Hardware Abstraction Layer. Outputs Hal.lib or Hal.dll.
* Kernel Program. Outputs Krnl32.lib or KRNL32.EXE

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、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

イェーイ！！！（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、付記されていた\*.hが削除され、すべてのCヘッダーの前にcが付記されます。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。

// Undefines NULL #ifdef NULL

# undef NULL #endif

#ifdef cplusplus extern "C"

{

#endif

/\* standard NULL declaration \*/ #define NULL 0

#ifdef cplusplus

}

#else

/\* standard NULL declaration \*/ #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

/\* standard size\_t type \*/ typedef unsigned size\_t;

#ifdef cplusplus

}

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タルの種類を見てみましょう。

typedef signed char int8\_t;

typedef unsigned char uint8\_t;

typedef short int16\_t; typedef unsigned short uint16\_t; typedef int int32\_t;

typedef unsigned uint32\_t;

typedef long long int64\_t; typedef unsigned long long uint64\_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。

#include <stdint.h> // C #include <cstdint> // C++ only

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。

ctype.h と cctype

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文字セットのさまざまなプロパティに従うことによって行われます。sciitable.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\_UP 0x01 /\* upper case \*/ #define CT\_LOW 0x02 /\* lower case \*/ #define CT\_DIG 0x04 /\* digit \*/ #define CT\_CTL 0x08 /\* control \*/ #define CT\_PUN 0x10 /\* punctuation \*/  #define CT\_WHT 0x20 /\* white space (space/cr/lf/tab) \*/ #define CT\_HEX 0x40 /\* hex digit \*/  #define CT\_SP 0x80 /\* hard space (0x20) \*/ | | | | | | |
| #define | isalnum(c) | ((\_ctype | + | 1)[(unsigned)(c)] | & | (CT\_UP | CT\_LOW | CT\_DIG)) |
| #define | isalpha(c) | ((\_ctype | + | 1)[(unsigned)(c)] | & | (CT\_UP | CT\_LOW)) |
| #define | iscntrl(c) | ((\_ctype | + | 1)[(unsigned)(c)] | & | (CT\_CTL)) |
| #define | isdigit(c) | ((\_ctype | + | 1)[(unsigned)(c)] | & | (CT\_DIG)) |
| #define | isgraph(c) | ((\_ctype | + | 1)[(unsigned)(c)] | & | (CT\_PUN | CT\_UP | CT\_LOW | CT\_DIG)) |
| #define | islower(c) | ((\_ctype | + | 1)[(unsigned)(c)] | & | (CT\_LOW)) |
| #define | isprint(c) | ((\_ctype | + | 1)[(unsigned)(c)] | & | (CT\_PUN | CT\_UP | CT\_LOW | CT\_DIG | CT\_SP)) |
| #define | ispunct(c) | ((\_ctype | + | 1)[(unsigned)(c)] | & | (CT\_PUN)) |
| #define | isspace(c) | ((\_ctype | + | 1)[(unsigned)(c)] | & | (CT\_WHT)) |
| #define | isupper(c) | ((\_ctype | + | 1)[(unsigned)(c)] | & | (CT\_UP)) |
| #define | isxdigit(c) | ((\_ctype | + | 1)[(unsigned)(c)] | & | (CT\_DIG | CT\_HEX)) |
| #define isascii(c) ((unsigned)(c) <= 0x7F) #define toascii(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、個々の文字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 list parameter list \*/ 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

/\* width of stack == width of int \*/ #define STACKITEM int

/\* round up width of objects pushed on stack. The expression before the & ensures that we get 0 for objects of size 0. \*/

#define VA\_SIZE(TYPE) \

((sizeof(TYPE) + sizeof(STACKITEM) - 1)

\ & ~(sizeof(STACKITEM) - 1))

これは少しやっかいです。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

/\* &(LASTARG) points to the LEFTMOST argument of the function call (before the ...) \*/

#define va\_start(AP, LASTARG)

\ (AP=((va\_list)&(LAS TARG) + 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

/\* nothing for va\_end \*/ #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

(AP += VA\_SIZE(TYPE), \*((TYPE \*)(AP - VA\_SIZE(TYPE))))

\

#define va\_arg(AP, TYPE)

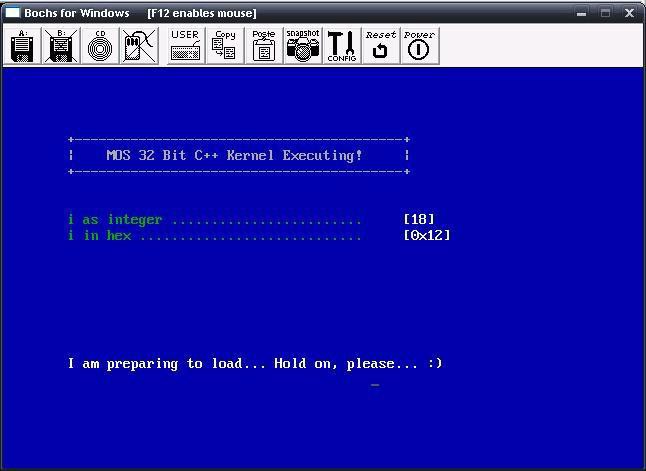
va\_arg()は，パラメータリストの次のパラメータを返しますが，これは少し厄介です．AP には、現在作業中のパラメータリストへのポインタを格納します。TYPEには、データ型（int、charなど）が入ります。

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ーは、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。

**Demo**

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 :)

Demo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N eptune Operating Systemを開発中 質問やコメントは？お気軽にご連絡ください。*

記事をより良くするために、あなたも貢献してみませんか？よろしければ、ぜひご連絡ください。



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。

Error Handling

Exception Handling

IRs, IRQs, ISRs

Gates: Traps, Interrupts, Tasks

IDTs and IVTs

IDTR processor register

LIDT and SIDT instructions

FLIHs and SLIHs

How interrupts work, stack, error codes

Developing a kernel panic error screen. ie, 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-、エラ-、エラ-。

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は、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、もう一つのフレーバーです。それは、現在の実行の流れを変える(?nterrupt?)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と同様です。

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。

割り込み処理

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気的干渉や、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

; generates software interrupt 3

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。

覚えておいてください。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チュートリアルの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と、8259A PICはプロセッサのIRライン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。

Byte 0: Offset Low Address of the Interrupt Routine (IR)

Byte 1: Offset High Address of the IR

Byte 2: Segment Low Address of the IR

Byte 3: Segment High Address of the IR

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 Interrupt Vector Table (IVT)** | | |
| Base Address | Interrupt Number | Description |
| 0x000 | 0 | Divide by 0 |
| 0x004 | 1 | Single step (Debugger) |
| 0x008 | 2 | Non Maskable Interrupt (NMI) Pin |
| 0x00C | 3 | Breakpoint (Debugger) |
| 0x010 | 4 | Overflow |
| 0x014 | 5 | Bounds check |
| 0x018 | 6 | Undefined Operation Code (OPCode) instruction |
| 0x01C | 7 | No coprocessor |
| 0x020 | 8 | Double Fault |
| 0x024 | 9 | Coprocessor Segment Overrun |
| 0x028 | 10 | Invalid Task State Segment (TSS) |
| 0x02C | 11 | Segment Not Present |
| 0x030 | 12 | Stack Segment Overrun |
| 0x034 | 13 | General Protection Fault (GPF) |
| 0x038 | 14 | Page Fault |
| 0x03C | 15 | Unassigned |
| 0x040 | 16 | Coprocessor error |
| 0x044 | 17 | Alignment Check (486+ Only) |
| 0x048 | 18 | Machine Check (Pentium/586+ Only) |
| 0x05C | 19-31 | Reserved exceptions |

|  |  |  |
| --- | --- | --- |
| 0x068 - 0x3FF | 32-255 | Interrupts free for software use |

難しいことではありません。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。そのため、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。

Bits 0...15:

**Interrupt / Trap Gate:** Offset address Bits 0-15 of IR

**Task Gate:**

Not used.

Bits 16...31:

**Interrupt / Trap Gate:** Segment Selector (Useually 0x10)

**Task Gate:** TSS Selector

Bits 31...35: Not used

Bits 36...38:

**Interrupt / Trap Gate:** Reserved. Must be 0.

**Task Gate:** Not used.

Bits 39...41:

**Interrupt Gate:** Of the format 0D110, where D determins size

**01110** - 32 bit descriptor

**00110** - 16 bit descriptor

**Task Gate:** Must be 00101

**Trap Gate:** Of the format 0D111, where D determins size

**01111** - 32 bit descriptor

**00111** - 16 bit descriptor

Bits 42...44: Descriptor Privedlge Level (DPL)

**00:** Ring 0

**01:** Ring 1

**10:** Ring 2

**11:** Ring 3

Bit 45: Segment is present (1: Present, 0:Not present)

Bits 46...62:

**Interrupt / Trap Gate:** Bits 16...31 of IR address

**Task Gate:** Not used

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:  .m\_baseLow | dw | 0 |
| .m\_selector | dw | 0x8 |
| .m\_reserved | db | 0 |
| .m\_flags | db | 010001110b |
| .m\_baseHi | dw | 0 |

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ではありませんよね。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レ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、コードセレ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レクタになります。

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。セレクタは常にGDT内のコードセレクタのもの（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

The IDTR register is the processor register that stores the base address of the IDT. The IDTR register has the following format:

|  |  |
| --- | --- |
| **IDTR Register** | |
| Bits 16...46 (IDT Base Address) | Bits 0...15 (IDT Limit) |

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\_ptr:

.limit dw idt\_end - idt\_start ; bits 0...15 is size of idt

.base dd idt\_start ; base of idt

; load register with idt\_ptr

そういえば、このレジスター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。

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。使用方法はとても簡単です。

lidt [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はディスクリプタのサイズ（ディスクリプタのサイズは8バイトであることを覚えていますか？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・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

1. ハンドラがより低い特権レベル（ディスクリプタのビット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。
2. ハンドラが使用するスタックのセグメント・セレクタとスタック・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・セレクタとスタック・ポインタをこの新しいスタックにプッシュします。
3. 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。

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。

1. 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
2. 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。

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る 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。

+---------------+ -- Bottom of stack

| EFLAGS |

+ +

| Return CS |

+ +

| Return EIP |

+ +

| Error Code |

+---------------+ -- ESP points here when handler is executed.

If there is no error code, ESP points to return EIP

この情報は、ハンドラーから戻ってきて、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・セレクタ・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・セレクタまたはゲート・セレクタ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。

ハンドラからの復帰

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) を復元することと、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 に設定されることが異なります。 また、CPL が IOPL 以下の場合にのみ IF フラグが変更されます。

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 Processor Exceptions** | | | |
| Interrupt Number | Class | Description | Error Code |
| 0 | Fault | Divide by 0 | None |
| 1 | Trap or Fault | Single step (Debugger) | None. Can be retrived from debug registers |
| 2 | Unclassed | Non Maskable Interrupt (NMI) Pin | Not applicable |
| 3 | Trap | Breakpoint (Debugger) | None |
| 4 | Trap | Overflow | None |
| 5 | Fault | Bounds check | None |
| 6 | Fault | Unvalid OPCode | None |
| 7 | Fault | Device not available | None |
| 8 | Abort | Double Fault | Always 0 |
| 9 | Abort (Reserved, do not use) | Coprocessor Segment Overrun | None |
| 10 | Fault | Invalid Task State Segment (TSS) | See error code below |
| 11 | Fault | Segment Not Present | See error code below |
| 12 | Fault | Stack Fault Exception | See error code below |
| 13 | Fault | General Protection Fault (GPF) | See error code below |
| 14 | Fault | Page Fault | See error code below |
| 15 | - | Unassigned | - |
| 16 | Fault | x87 FPU Error | None. x87 FPU provides own error information |
| 17 | Fault | Alignment Check (486+ Only) | Always 0 |
| 18 | Abort | Machine Check (Pentium/586+ Only) | None. Error information abtained from MSRs |
| 19 | Fault | SIMD FPU Exception | None |
| 20-31 | - | Reserved | - |
| 32-255 | - | Avilable for software use | Not applicable |

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？ビンゴです。

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てしまったので、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きますか？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）のリマッピング

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こ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。

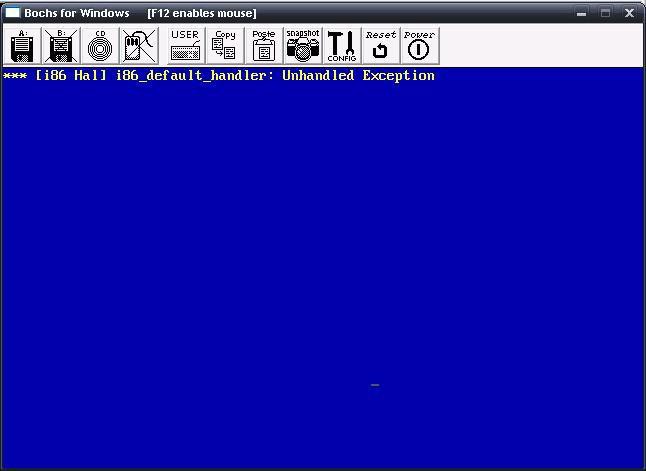
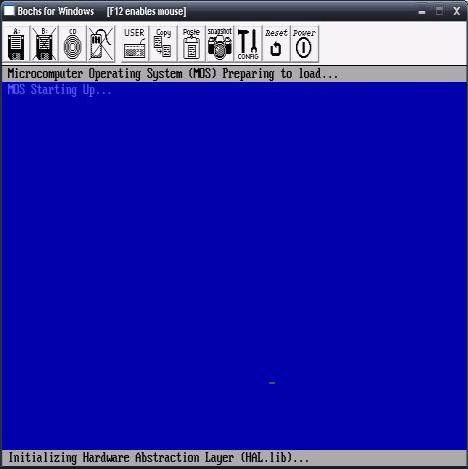
8259A PICは、プログラミングがかなり複雑なマイクロコントローラです。幸いなことに、そのモードのほとんどは私たちには当てはまりません。

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、8 259A Programmable Interrupt Controllerのチュートリアルを読むことをお勧めします。

**Demo**

デモの結論に至るまでの過程

*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に注目してください。*

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ースを 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。

楽しみですね :)

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で、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ースの始まりをご紹介しましょう。

Hal - include/hal.h - プラットフォームに依存しないHAL用インターフェース

これは、HALとカーネルの間のインターフェースです。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。

//! Initialize and shutdown hal extern int Hal\_Initialize (); extern int Hal\_Shutdown ();

おそらく、これらのルーチンのプロトタイプを変更して、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すると思います。

いずれにしても、これら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ています。

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の初期化ルーチンを呼び出し、それがgdtとidtの初期化メソッドを呼び出す)、このチュートリアルに必要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 。

**Bits 56-63:** Bits 24-32 of the base address

**Bit 55:** Granularity

**0:** None

**1:** Limit gets multiplied by 4K

**Bit 54:** Segment type

**0:** 16 bit

**1:** 32 bit

**Bit 53:** Reserved-Should be zero

**Bits 52:** Reserved for OS use

**Bits 48-51:** Bits 16-19 of the segment limit

**Bit 47** Segment is in memory (Used with Virtual Memory)

**Bits 45-46:** Descriptor Privilege Level

**0:** (Ring 0) Highest

**3:** (Ring 3) Lowest

**Bit 44:** Descriptor Bit

**0:** System Descriptor

**1:** Code or Data Descriptor

**Bits 41-43:** Descriptor Type

**Bit 43:** Executable segment

**0:** Data Segment

**1:** Code Segment

**Bit 42:** Expansion direction (Data segments), conforming (Code Segments)

**Bit 41:** Readable and Writable

**0:** Read only (Data Segments); Execute only (Code Segments)

**1:** Read and write (Data Segments); Read and Execute (Code Segments)

**Bit 40:** Access bit (Used with Virtual Memory)

**Bits 16-39:** Bits 0-23 of the Base Address

**Bits 0-15:** Bits 0-15 of the Segment Limit

\*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で、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、フラグの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。

#ifdef \_MSC\_VER #pragma pack (push, 1) #endif

//! gdt descriptor. A gdt descriptor defines the properties of a specific

//! memory block and permissions. struct gdt\_descriptor {

//! bits 0-15 of segment limit uint16\_t limit;

//! bits 0-23 of base address uint16\_t baseLo;

uint8\_t baseMid;

//! descriptor bit flags. Set using bit masks above uint16\_t flags;

//! bits 24-32 of base address uint8\_t baseHi;

};

#ifdef \_MSC\_VER #pragma pack (pop, 1) #endif

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。

gdtrの抽象化

チュートリアル8では、プロテクトモード、gdt、gdtrについて説明しました。 gdtr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、次のような形式で指定します。

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。

#ifdef \_MSC\_VER #pragma pack (push, 1) #endif

//! processor gdtr register points to base of gdt. This helps

//! us set up the pointer struct gdtr {

//! size of gdt uint16\_t

m\_limit;

//! base address of gdt uint32\_t m\_base;

};

#ifdef \_MSC\_VER #pragma pack (pop, 1) #endif

// Global Descriptor Table (GDT) static struct gdt\_descriptor

\_gdt [MAX\_DESCRIPTORS];

//! gdtr data static struct gdtr

\_gdtr;

このページでは、新しいGDTと\_gdtr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gdtr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にgdtrポインタをロードするだけです。CSが変化しないように、ここではファージャンプをする必要はありません。

//! installs gdtr

static void gdt\_install ()

{ #ifdef \_MSC\_VER

\_asm lgdt [\_gdtr]

#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。

//! Setup a descriptor in the Global Descriptor Table

void gdt\_set\_descriptor(uint32\_t i, uint64\_t base, uint64\_t limit, uint8\_t access, uint8\_t grand)

{

if (i > MAX\_DESCRIPTORS)

return;

//! null out the descriptor

memset ((void\*)&\_gdt[i], 0, sizeof (gdt\_descriptor));

//! set limit and base addresses

\_gdt[i].baseLo = base & 0xffff;

\_gdt[i].baseMid = (base >> 16) & 0xff;

\_gdt[i].baseHi = (base >> 24) & 0xff;

\_gdt[i].limit = limit & 0xffff;

//! set flags and grandularity bytes

\_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(0xffffff)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。

//! initialize gdt

int i86\_gdt\_initialize () {

//! set up gdtr

\_gdtr.m\_limit = (sizeof (struct gdt\_descriptor) \* MAX\_DESCRIPTORS)-1;

\_gdtr.m\_base = (uint32\_t)&\_gdt[0];

//! set null descriptor gdt\_set\_descriptor(0, 0, 0, 0, 0);

//! set default code descriptor 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);

//! set default data descriptor 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);

//! install gdtr gdt\_install ();

return 0;

}

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。

#ifdef \_MSC\_VER #pragma pack (push, 1) #endif

//! interrupt descriptor struct idt\_descriptor {

//! bits 0-16 of interrupt routine (ir) address uint16\_t baseLo;

//! code selector in gdt uint16\_t sel;

//! reserved, shold be 0

uint8\_t reserved;

//! bit flags. Set with flags above uint8\_t flags;

//! bits 16-32 of ir address uint16\_t baseHi;

};

#ifdef \_MSC\_VER #pragma pack (pop, 1) #endif

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。構造

sel - セグメントセレクター

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31～38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、ビットフラグが

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。

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46～64です。

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。

#ifdef \_MSC\_VER #pragma pack (push, 1) #endif

//! describes the structure for the processors idtr register struct idtr {

//! size of the interrupt descriptor table (idt) uint16\_t limit;

//! base address of idt uint32\_t base;

};

#ifdef \_MSC\_VER #pragma pack (pop, 1) #endif

//! interrupt descriptor table

static struct idt\_descriptor \_idt [I86\_MAX\_INTERRUPTS];

//! idtr structure used to help define the cpu's idtr register static struct idtr \_idtr;

なるほど...。IDTは割り込み記述子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。難しいことではありません。）

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。

//! installs idtr into processors 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インターフェースは、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。

//! default handler to catch unhandled system interrupts. void i86\_default\_handler () {

#ifdef \_DEBUG

DebugClrScr (0x18); DebugGotoXY (0,0); DebugSetColor (0x1e);

DebugPrintf ("\*\*\* [i86 Hal] i86\_default\_handler: Unhandled Exception");

#endif

for(;;);

}

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 abritary interrupt call を生成するという課題も隠されています。

問題は、割り込み（INT命令）のOPCodeのフォーマットが1つしかないことです。0xCDimmで、immは中間値です。このため、INT命令ではレジスタやメモリロケーションを使用することができません。これを受け入れるOPCode形式がないからです（無効な命令です）。もちろん、さまざまな方法があります。私は、早くて小さな解決策として、自己修正コードを使うことにしました。

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。

//! generate interrupt call void geninterrupt (int n)

{ #ifdef \_MSC\_VER

\_asm {

mov al, byte ptr [n]

mov byte ptr [genint+1], al jmp genint

genint:

int 0 // above code modifies the 0 to int number to generate

}

#endif

}

i86\_install\_ir () - 割り込みハンドラをIDTにインストールする

これは少し厄介ですが、それほど難しくはありません。構造体のbaseLoとbaseHiには、割り込みルーチン(IR)の上位ビットと下位ビットが格納されて いることを覚えていますか？つまり、IR関数のアドレスを取得して、その上位ビットと下位ビットを格納すればいいのです。これは、関数ポインタを使って行います。

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 [i]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。

//! installs a new interrupt handler

int i86\_install\_ir (uint32\_t i, uint16\_t flags, uint16\_t sel, I86\_IRQ\_HANDLER irq) {

if (i>I86\_MAX\_INTERRUPTS)

return 0;

if (!irq)

return 0;

//! get base address of interrupt handler

uint64\_t uiBase = (uint64\_t)&(\*irq);

//! store base address into idt

|  |  |  |
| --- | --- | --- |
| \_idt[i].baseLo | = | uiBase & 0xffff; |
| \_idt[i].baseHi | = | (uiBase >> 16) & 0xffff; |
| \_idt[i].reserved | = | 0; |
| \_idt[i].flags | = | flags; |
| \_idt[i].sel | = | sel; |
| return 0;  } |  |  |

これにはいくつかの理由があります。割り込みが発生すると、プロセッサは私たちのためにいくつかの情報をスタックにプッシュすることを覚えていますか？この情報は、私たちのルーチンが呼ばれたときに、パラメタリストに入ります。すごいでしょう？ただし、エラーコードをプッシュする割り込みとしない割り込みがありますので、注意が必要です。

i86\_idt\_initialize () - IDTインターフェースの初期化

さて、すべてをまとめてみましょう。次のコードは、IDTRを設定し、すべての割り込みをキャッチするデフォルトの割り込みハンドラを設定し（これにより、カーネルで必要な割り込みを定義するだけで済むようになります）、最後に上記の方法でIDTをインストールしています。

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。

//! initialize idt

int i86\_idt\_initialize (uint16\_t codeSel) {

//! set up idtr for processor

\_idtr.limit = sizeof (struct idt\_descriptor) \* I86\_MAX\_INTERRUPTS -1;

\_idtr.base = (uint32\_t)&\_idt[0];

//! null out the idt

memset ((void\*)&\_idt[0], 0, sizeof (idt\_descriptor) \* I86\_MAX\_INTERRUPTS-1);

//! register default handlers

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);

//! install our idt idt\_install ();

return 0;

}

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、 正しい割込みハンドラ（または割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

//! our uber 1337 interrupt handler. handles int 5 request void int\_handler\_5 () {

\_asm add esp, 12

\_asm pushad

// do whatever...

\_asm popad

\_asm iretd

}

//! registers our interrupt handler

i86\_install\_ir (5, I86\_IDT\_DESC\_PRESENT | I86\_IDT\_DESC\_BIT32, 0x8, (I86\_IRQ\_HANDLER)int\_handler\_5);

//! generates int 5 instruction. You can also use inline assembly, of course geninterrupt (5);

割り込みハンドラのパラメタリストはフォーマットが変わる可能性があるので、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。後でパラメタを与えて楽にしようと思うかもしれませんが。

Demo Download Here (MSVC+)

結論

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、例外処理と割込み処理を説明し、システムの割込みを再び有効にしてきました。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？^\_^

In the next tutorial, we will be starting to develop our kernel even more. We will be handling timing through the **8254 Programmable Interval Timer (PIT) microcontroller**, which will be covered simular to the 8259A PIC tutorial. Afterwords, we plan on moving onto more memory management and process management. .. We might even develop a basic debugging text based console to spice things を少しアップしました。）

次の機会まで。

~マイク

*BrokenThorn Entertainment社。現在、DoEとN eptune Operating Systemを開発中 質問やコメントは？お気軽にご連絡ください。*

記事をより良くするために、あなたも貢献してみませんか？よろしければ、ぜひご連絡ください。



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について説明しており、ソフトウェア割り込みの処理については説明していません。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？

In the last tutorial, we have dived deep into the world of interrupt handling. We have covered, and even implimented, interfaces for the GDT and IDT inside of our **hardware abstraction layer**. We have covered almost everything we needed for software interrupt handling to work But WAIT! What

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。

Hardware Interrupts

Interrupt Chaining

Hal: Programmable Interrupt Controller Hal: Programmable Interval Timer Hardware Abstraction

Interrupts Implimentation and Design for our HAL

なお、ここでは割り込み処理については説明していません。割り込み処理についてはチュートリアル15をご覧ください。それを踏まえた上で、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 Hardware Interrupts** | | |
| 8259A Input pin | Interrupt Number | Description |
| IRQ0 | 0x08 | Timer |
| IRQ1 | 0x09 | Keyboard |
| IRQ2 | 0x0A | Cascade for 8259A Slave controller |
| IRQ3 | 0x0B | Serial port 2 |
| IRQ4 | 0x0C | Serial port 1 |
| IRQ5 | 0x0D | AT systems: Parallel Port 2. PS/2 systems: reserved |
| IRQ6 | 0x0E | Diskette drive |
| IRQ7 | 0x0F | Parallel Port 1 |
| IRQ8/IRQ0 | 0x70 | CMOS Real time clock |
| IRQ9/IRQ1 | 0x71 | CGA vertical retrace |
| IRQ10/IRQ2 | 0x72 | Reserved |
| IRQ11/IRQ3 | 0x73 | Reserved |
| IRQ12/IRQ4 | 0x74 | AT systems: reserved. PS/2: auxiliary device |
| IRQ13/IRQ5 | 0x75 | FPU |
| IRQ14/IRQ6 | 0x76 | Hard disk controller |
| IRQ15/IRQ7 | 0x77 | Reserved |

各デバイスについては、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。

Wait. .What is this PIC thing? All of these hardware devices that can signal hardware devices are connected indirectly to the **8259A Programmable**

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ー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少し後に行います。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ます。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。

割り込みチャーンとは、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。

void deviceInitialize () {

//store previus interrupt handler prevhandler = getvect (0);

//install new interrupt handler setvect (0, handler);

}

void deviceShutdown () {

//install previus interrupt handler setvect (0, prevhandler);

}

void handler () {

// do stuff...

// call previus interrupt handler (\*prevhandler) ();

}

ご覧のように、割り込みの連鎖はとても簡単です。setvect()は新しい割込みベクターを設置し、getvect()は割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、ここまで来るに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、ハドウェア割り込みを再び有効にするとすぐに、次のタイマーチ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を再プログラムする方法も学ば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、読者の皆さんには、次のチュートリアルを読んでから続けることをお勧めします。8 259 プロ

グラム可能なインタラプトコントローラ

8 253 プログラム可能なインターバルタイマ

これらのチュートリアルの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の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、あなたを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いただくとよいでしょう。

So..? What are you waiting for? Jump into those tutorials! And come back here when you are done. Don't worry, I am just text... I will still be here when you get back unless I am deleted of course :)

**Demo**



*新デモの動作*

Demo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。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| extern | int |  | \_cdecl | hal\_initialize (); |
| extern | int |  | \_cdecl | hal\_shutdown (); |
| extern | void |  | \_cdecl | enable (); |
| extern | void |  | \_cdecl | disable (); |
| extern | void |  | \_cdecl | geninterrupt (int n); |
| extern | unsigned char |  | \_cdecl | inportb (unsigned short id); |
| extern | void |  | \_cdecl | outportb (unsigned short id, unsigned char value); |
| extern | void |  | \_cdecl | setvect (int intno, void (\_cdecl far &vect) ( ) ); |
| extern | void (\_cdecl | far \* | \_cdecl | getvect (int intno)) ( ); |
| extern | bool |  | \_cdecl | interruptmask (uint8\_t intno, bool enable); |
| extern | inline void |  | \_cdecl | interruptdone (unsigned int intno); |
| extern | void |  | \_cdecl | sound (unsigned frequency); |
| extern | const char\* |  | \_cdecl | get\_cpu\_vender (); |
| extern | int |  | \_cdecl | get\_tick\_count (); |

16bit DOSをプログラムしたことがある人は、今すぐにで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している必要があります。

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。

**PIC:**

8プ259A 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ー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。

これの一番の利点は、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イスを提供します。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。覚えておいてください。魔法の数字は悪いものです。

|  |  |  |
| --- | --- | --- |
| //! The | following devices use PIC 1 to generate | interrupts |
| #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 |
| //! The | following devices use PIC 2 to generate | interrupts |
| #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ではありません。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すか？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。

|  |  |  |
| --- | --- | --- |
| **Operation Command Word (OCW) 2** | | |
| Bit Number | Value | Description |
| 0-2 | L0/L1/L2 | Interrupt level upon which the controller must react |

|  |  |  |
| --- | --- | --- |
| 3-4 | 0 | Reserved, must be 0 |
| 5 | EOI | End of Interrupt (EOI) request |
| 6 | SL | Selection |
| 7 | R | Rotation option |

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。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| //! Command Word 2 bit masks. Use when sending commands | | | | |
| #define | I86\_PIC\_OCW2\_MASK\_L1 | 1 | //00000001 | //Level 1 interrupt level |
| #define | I86\_PIC\_OCW2\_MASK\_L2 | 2 | //00000010 | //Level 2 interrupt level |
| #define | I86\_PIC\_OCW2\_MASK\_L3 | 4 | //00000100 | //Level 3 interrupt level |
| #define | I86\_PIC\_OCW2\_MASK\_EOI | 0x20 | //00100000 | //End of Interrupt command |
| #define | I86\_PIC\_OCW2\_MASK\_SL | 0x40 | //01000000 | //Select command |
| #define | I86\_PIC\_OCW2\_MASK\_ROTATE | 0x80 | //10000000 | //Rotation command |

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 bit masks. Use when sending commands | | | |
| #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 #define | 1 register port addresses I86\_PIC1\_REG\_COMMAND | 0x20 | // command register |
| #define | I86\_PIC1\_REG\_STATUS | 0x20 | // status register |
| #define | I86\_PIC1\_REG\_DATA | 0x21 | // data register |
| #define | I86\_PIC1\_REG\_IMR | 0x21 | // interrupt mask register (imr) |
| //! PIC #define | 2 register port addresses I86\_PIC2\_REG\_COMMAND | 0xA0 | // ^ see above register names |
| #define #define #define | I86\_PIC2\_REG\_STATUS I86\_PIC2\_REG\_DATA I86\_PIC2\_REG\_IMR | 0xA0  0xA1  0xA1 |  |

難しいことではありません。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、コマンドレジスター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

これは、PICを初期化する際に使用する一次制御ワードです。これは7ビットの値で、プライマリPICコマンドレジスタに入れなければなりません。これは、フォーマットです。

|  |  |  |
| --- | --- | --- |
| **Initialization Control Word (ICW) 1** | | |
| Bit Number | Value | Description |
| 0 | IC4 | If set(1), the PIC expects to recieve IC4 during initialization. |
| 1 | SNGL | If set(1), only one PIC in system. If cleared, PIC is cascaded with slave PICs, and ICW3 must be sent to controller. |
| 2 | ADI | If set (1), CALL address interval is 4, else 8. This is useually ignored by x86, and is default to 0 |
| 3 | LTIM | If set (1), Operate in Level Triggered Mode. If Not set (0), Operate in Edge Triggered Mode |
| 4 | 1 | Initialization bit. Set 1 if PIC is to be initialized |
| 5 | 0 | MCS-80/85: Interrupt Vector Address. x86 Architecture: Must be 0 |
| 6 | 0 | MCS-80/85: Interrupt Vector Address. x86 Architecture: Must be 0 |
| 7 | 0 | MCS-80/85: Interrupt Vector Address. x86 Architecture: Must be 0 |

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。

|  |  |  |  |
| --- | --- | --- | --- |
| //! Initialization Control Word 1 bit masks | | | |
| #define I86\_PIC\_ICW1\_MASK\_IC4 | 0x1 | //00000001 | // Expect ICW 4 bit |
| #define I86\_PIC\_ICW1\_MASK\_SNGL | 0x2 | //00000010 | // Single or Cascaded |
| #define I86\_PIC\_ICW1\_MASK\_ADI | 0x4 | //00000100 | // Call Address Interval |
| #define I86\_PIC\_ICW1\_MASK\_LTIM | 0x8 | //00001000 | // Operation Mode |
| #define I86\_PIC\_ICW1\_MASK\_INIT | 0x10 | //00010000 | // Initialization Command |

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。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| #define | I86\_PIC\_ICW1\_IC4\_EXPECT | 1 | //1 | //Use when setting I86\_PIC\_ICW1\_MASK\_IC4 |
| #define | I86\_PIC\_ICW1\_IC4\_NO | 0 | //0 |  |
| #define | I86\_PIC\_ICW1\_SNGL\_YES | 2 | //10 | //Use when setting I86\_PIC\_ICW1\_MASK\_SNGL |
| #define | I86\_PIC\_ICW1\_SNGL\_NO | 0 | //00 |  |
| #define | I86\_PIC\_ICW1\_ADI\_CALLINTERVAL4 | 4 | //100 | //Use when setting I86\_PIC\_ICW1\_MASK\_ADI |
| #define | I86\_PIC\_ICW1\_ADI\_CALLINTERVAL8 | 0 | //000 |  |
| #define | I86\_PIC\_ICW1\_LTIM\_LEVELTRIGGERED | 8 | //1000 | //Use when setting I86\_PIC\_ICW1\_MASK\_LTIM |
| #define | I86\_PIC\_ICW1\_LTIM\_EDGETRIGGERED | 0 | //0000 |  |
| #define | I86\_PIC\_ICW1\_INIT\_YES | 0x10 | //10000 | //Use when setting I86\_PIC\_ICW1\_MASK\_INIT |
| #define | I86\_PIC\_ICW1\_INIT\_NO | 0 | //00000 |  |

難しいことではありません。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。

uint8\_t icw=0;

icw = (icw & ~I86\_PIC\_ICW1\_MASK\_INIT) | I86\_PIC\_ICW1\_INIT\_YES; icw = (icw & ~I86\_PIC\_ICW1\_MASK\_IC4) | I86\_PIC\_ICW1\_IC4\_EXPECT;

そうなんだ！？そうです。すべてがうまく機能し、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、さまざまな場面で使用されています。このコードを見ただけで、何をしているのかがわ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）。かっこいいでしょう？このシリーズでは、ビットの設定やマスキングの際に、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。

|  |  |  |
| --- | --- | --- |
| **Initialization Control Word (ICW) 2** | | |
| Bit Number | Value | Description |
| 0-2 | A8/A9/A10 | Address bits A8-A10 for IVT when in MCS-80/85 mode. |
| 3-7 | A11(T3)/A12(T4)/A13(T5)/A14(T6)/A15(T7) | Address bits A11-A15 for IVT when in MCS-80/85 mode. **In 80x86 mode, specifies the interrupt vector address.** May be set to 0 in x86 mode. |

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コントローラー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します。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。

|  |  |  |
| --- | --- | --- |
| **Initialization Control Word (ICW) 3** | | |
| Bit Number | Value | Description |
| 0-7 | S0-S7 | Specifies what Interrupt Request (IRQ) is connected to slave PIC |

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

イェーイ! 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。

|  |  |  |
| --- | --- | --- |
| **Initialization Control Word (ICW) 4** | | |
| Bit Number | Value | Description |
| 0 | uPM | If set (1), it is in 80x86 mode. Cleared if MCS-80/86 mode |
| 1 | AEOI | If set, on the last interrupt acknowledge pulse, controller automatically performs End of Interrupt (EOI) operation |
| 2 | M/S | Only use if BUF is set. If set (1), selects buffer master. Cleared if buffer slave. |
| 3 | BUF | If set, controller operates in buffered mode |
| 4 | SFNM | Special Fully Nested Mode. Used in systems with a large amount of cascaded controllers. |
| 5-7 | 0 | Reserved, must be 0 |

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。

|  |  |  |  |
| --- | --- | --- | --- |
| //! Initialization Control Word 4 bit masks | | | |
| #define I86\_PIC\_ICW4\_MASK\_UPM | 0x1 | //00000001 | // Mode |
| #define I86\_PIC\_ICW4\_MASK\_AEOI | 0x2 | //00000010 | // Automatic EOI |
| #define I86\_PIC\_ICW4\_MASK\_MS | 0x4 | //00000100 | // Selects buffer type |
| #define I86\_PIC\_ICW4\_MASK\_BUF | 0x8 | //00001000 | // Buffered mode |
| #define I86\_PIC\_ICW4\_MASK\_SFNM | 0x10 | //00010000 | // Special fully-nested mode |

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て使用され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 | //Use when setting I86\_PIC\_ICW4\_MASK\_UPM |
| #define | I86\_PIC\_ICW4\_UPM\_MCSMODE | 0 | //0 |  |
| #define | I86\_PIC\_ICW4\_AEOI\_AUTOEOI | 2 | //10 | //Use when setting I86\_PIC\_ICW4\_MASK\_AEOI |
| #define | I86\_PIC\_ICW4\_AEOI\_NOAUTOEOI | 0 | //00 |  |
| #define | I86\_PIC\_ICW4\_MS\_BUFFERMASTER | 4 | //100 | //Use when setting I86\_PIC\_ICW4\_MASK\_MS |
| #define | I86\_PIC\_ICW4\_MS\_BUFFERSLAVE | 0 | //000 |  |
| #define | I86\_PIC\_ICW4\_BUF\_MODEYES | 8 | //1000 | //Use when setting I86\_PIC\_ICW4\_MASK\_BUF |
| #define | I86\_PIC\_ICW4\_BUF\_MODENO | 0 | //0000 |  |
| #define | I86\_PIC\_ICW4\_SFNM\_NESTEDMODE | 0x10 | //10000 | //Use when setting I86\_PIC\_ICW4\_MASK\_SFNM |
| #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て、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、関数をご紹介しましょう。

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で必要となります。

inline void i86\_pic\_send\_command (uint8\_t cmd, uint8\_t picNum)

{ if (picNum > 1)

return;

uint8\_t reg = (picNum==1) ? I86\_PIC2\_REG\_COMMAND : I86\_PIC1\_REG\_COMMAND; outportb (reg, cmd);

}

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よく似ていますが、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。

inline void i86\_pic\_send\_data (uint8\_t data, uint8\_t picNum) {

if (picNum > 1)

return;

uint8\_t reg = (picNum==1) ? I86\_PIC2\_REG\_DATA : I86\_PIC1\_REG\_DATA; outportb (reg, data);

}

inline uint8\_t i86\_pic\_read\_data (uint8\_t picNum)

{ if (picNum > 1)

return 0;

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。また、I86\_PIC\_ICW1\_IC4\_EXPECTビットを設定して、コントローラがICW4を送信することを期待していることを確認しています。

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。

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P ICチュートリアルの例と比較してみて、その類似性に驚いてください。

}

//! Begin initialization of PIC

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);

//! Send initialization control word 2. This is the base addresses of the irq's i86\_pic\_send\_data (base0, 0);

i86\_pic\_send\_data (base1, 1);

//! Send initialization control word 3. This is the connection between master and slave.

//! ICW3 for master PIC is the IR that connects to secondary pic in binary format

//! ICW3 for secondary PIC is the IR that connects to master pic in decimal format

i86\_pic\_send\_data (0x04, 0);

i86\_pic\_send\_data (0x02, 1);

//! Send Initialization control word 4. Enables i86 mode

icw = (icw & ~I86\_PIC\_ICW4\_MASK\_UPM) | I86\_PIC\_ICW4\_UPM\_86MODE;

i86\_pic\_send\_data (icw, 0);

i86\_pic\_send\_data (icw, 1);

= 0;

icw

uint8\_t

//! Initialize pic

void i86\_pic\_initialize (uint8\_t base0, uint8\_t base1) {

\*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IR0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するつもりはありません。

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。

**Bit 0: (BCP)** Binary Counter

**0:** Binary

**1:** Binary Coded Decimal (BCD)

**Bit 1-3: (M0, M1, M2)** Operating Mode. See above sections for a description of each.

**000:** Mode 0: Interrupt or Terminal Count

**001:** Mode 1: Programmable one-shot

**010:** Mode 2: Rate Generator

**011:** Mode 3: Square Wave Generator

**100:** Mode 4: Software Triggered Strobe

**101:** Mode 5: Hardware Triggered Strobe

**110:** Undefined; Don't use

**111:** Undefined; Don't use

**Bits 4-5: (RL0, RL1)** Read/Load Mode. We are going to read or send data to a counter register

**00:** Counter value is latched into an internal control register at the time of the I/O write operation.

**01:** Read or Load Least Significant Byte (LSB) only **10:** Read or Load Most Significant Byte (MSB) only **11:** Read or Load LSB first then MSB

**Bits 6-7: (SC0-SC1)** Select Counter. See above sections for a description of each.

**00:** Counter 0

**01:** Counter 1

**10:** Counter 2

**11:** Illegal value

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 |

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 | //! Use when setting I86\_PIT\_OCW\_MASK\_BINCOUNT |
| #define | I86\_PIT\_OCW\_BINCOUNT\_BCD | 1 | //1 |  |
| #define | I86\_PIT\_OCW\_MODE\_TERMINALCOUNT | 0 | //0000 | //! Use when setting I86\_PIT\_OCW\_MASK\_MODE |
| #define | I86\_PIT\_OCW\_MODE\_ONESHOT | 0x2 | //0010 |  |
| #define | I86\_PIT\_OCW\_MODE\_RATEGEN | 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 | //! Use when setting I86\_PIT\_OCW\_MASK\_RL |
| #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 | //! Use when setting I86\_PIT\_OCW\_MASK\_COUNTER |
| #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。うわぁぁぁぁぁぁぁぁぁぁぁぁぁぁぁぁぁぁぁぁぁぁぁぁぁぁ

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 |
| //! Global Tick count  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、各カウンターのデータレジスタです。PITには3 つの内部カウンター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IR0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。

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用することができます。

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。

void interrupt \_cdecl i86\_pit\_irq () {

//! macro to hide interrupt start code intstart ();

//! increment tick count

\_pit\_ticks++;

//! tell hal we are done interruptdone(0);

//! macro used with intstart to return from interrupt handler intret ();

}

i86\_pit\_send\_command ()です。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非常に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。コマンドはOCW(Operation Command Word)という形式で送られます。

//! send command to pic

void i86\_pit\_send\_command (uint8\_t cmd) {

outportb (I86\_PIT\_REG\_COMMAND, cmd);

}

例えば、上記のビットマスクと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に送ることができる。

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のに役立ちます。これらのルーチンは、現在のカウント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、正しいポートを取得す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。

//! send data to a counter

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);

outportb (port, data);

}

//! read data from counter

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);

return inportb (port);

}

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）のコードセレクタのオフセットです。

i86\_install\_ir()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。

//! initialize minidriver

void i86\_pit\_initialize (uint8\_t irq, uint8\_t irCodeSeg) {

//! Install our interrupt handler

i86\_install\_ir (irq, I86\_IDT\_DESC\_PRESENT | I86\_IDT\_DESC\_BIT32, irCodeSeg, i86\_pit\_irq);

}

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\_PIT\_OCW\_MODE\_SQUAREWAVEGEN など)。 freq には、カウンタに動作させたい周波数レートを指定します。

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)

return;

uint16\_t divisor = 1193180 / freq;

//! send operational command 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);

//! set frequency rate i86\_pit\_send\_data (divisor & 0xff, 0);

i86\_pit\_send\_data ((divisor >> 8) & 0xff, 0);

//! reset tick count

\_pit\_ticks=0;

}

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N eptune Operating Systemを開発中 質問やコメントは？お気軽にご連絡ください。*

記事をより良くするために、あなたも貢献してみませんか？よろしければ、ぜひご連絡ください。



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 - 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。

Physical Memory

Translation Lookaside Buffer (TLB) Memory Management Unit (MMU) Memory Managers

Abtaining memory information

Passing information from bootloader to kernel Designing and developing a physical memory manager

それでは、アリレイティ 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るのでしょうか？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章で説明しましたね）。

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的な機能です。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。

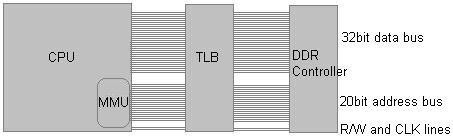
第4章ではセグメント：オフセットのアドレッシングモードを、第8章ではディスクリプター：オフセットのメモリアドレッシングモードを取り上げました。

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、実際に何が起こるのでしょうか。書き込みの際には、プロセッサーは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と、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を使えないんじゃなかったっけ？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ご紹介しますが、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最後にある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)

**Return**

CF = Clear if successful

AX = Number of KB conventional memory

AH = status if error (0x80: Invalid command; 0x86) Unsupported function

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。

ax, -1

mov ret

.error:

; if size=0

;unsupported function

;invalid command

0x12

.error ax, ax

.error ah, 0x86

.error ah, 0x80

.error

int jc test je cmp je cmp je ret

BiosGetMemorySize:

BIOS INT 0x15 Function 0x88 - Get Extended Memory Size

**Return**

CF = Clear if successful

AX = Number of contiguous KB starting at 1MB physical address

AH = status if error (0x80: Invalid command; 0x86) Unsupported function

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: | | |
| mov | ax, 0x88 |  |
| int | 0x15 |  |
| jc | .error |  |
| test | ax, ax | ; if size=0 |
| je | .error |  |
| cmp | ah, 0x86 | ;unsupported function |
| je | .error |  |
| cmp | ah, 0x80 | ;invalid command |
| je | .error |  |
| ret |  |  |
| .error: |  |  |
| mov | ax, -1 |  |
| ret |  |  |

BIOS INT 0x15 Function 0xE881 - Get Memory Size For > 64 MB Configuations (32 Bit)

**Return**

CF = Clear if successful

EAX = Extended memory between 1MB and 16MB in KB EBX = Extended memory above 16MB, in 64KB blocks ECX = Configured memory 1MB to 16MB in KB

EDX = Configured memory above 16MB in 64JB blocks

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 Configuations

**Return**

CF = Clear if successful

EAX = Extended memory between 1MB and 16MB in KB EBX = Extended memory above 16MB, in 64KB blocks ECX = Configured memory 1MB to 16MB in KB

EDX = Configured memory above 16MB in 64JB blocks

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。

;---------------------------------------------

; Get memory size for >64M configuations

; ret\ ax=KB between 1MB and 16MB

; ret\ bx=number of 64K blocks above 16MB

; ret\ bx=0 and ax= -1 on error

;---------------------------------------------

BiosGetMemorySize64MB:

.use\_ax:

.error:

mov mov pop pop ret

ax, -1

bx, 0 edx ecx

|  |  |  |
| --- | --- | --- |
| push push xor  xor | ecx edx  ecx, ecx  edx, edx | ;clear all registers. This is needed for testing later |
| mov | ax, 0xe801 |  |
| int | 0x15 |  |
| jc | .error |  |
| cmp | ah, 0x86 | ;unsupported function |
| je | .error |  |
| cmp | ah, 0x80 | ;invalid command |
| je | .error |  |
| jcxz | .use\_ax | ;bios may have stored it in ax,bx or cx,dx. test if cx is 0 |
| mov | ax, cx | ;its not, so it should contain mem size; store it |
| mov | bx, dx |  |

|  |  |  |
| --- | --- | --- |
| pop | edx | ;mem size is in ax and bx already, return it |
| 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が、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、データバ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よね。

そこで登場するのが「メモリーマップ」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

**Input**

EAX = 0x0000E820

EBX = continuation value or 0 to start at beginning of map

ECX = size of buffer for result (Must be >= 20 bytes)

EDX = 0x534D4150h ('SMAP')

ES:DI = Buffer for result

**Return**

CF = clear if successful

EAX = 0x534D4150h ('SMAP')

EBX = offset of next entry to copy from or 0 if done

ECX = actual length returned in bytes

ES:DI = buffer filled

If error, AH containes error code

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されるバッファは、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。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| struc | MemoryMapEntry |  |  |  |
|  | .baseAddress | resq | 1 | ; base address of address range |
|  | .length | resq | 1 | ; length of address range in bytes |
|  | .type | resd | 1 | ; type of address range |
|  | .acpi\_null | resd | 1 | ; reserved |
| endstruc | |  |  |  |

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

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。

;---------------------------------------------

; Get memory map from bios

; /in es:di->destination buffer for entries

; /ret bp=entry count

;---------------------------------------------

BiosGetMemoryMap:

pushad

.next\_entry:

.start:

.notext:

mov test jne mov jecxz

.good\_entry:

ecx, [es:di + MemoryMapEntry.length]

; get length (low dword)

ecx, ecx

short .good\_entry

; if length is 0 skip it

ecx, [es:di + MemoryMapEntry.length + 4]; get length (upper dword)

.skip\_entry

; if length is 0 skip it

.skip\_entry:

.error:

|  |  |  |  |
| --- | --- | --- | --- |
| xor  xor | ebx, ebx  bp, bp | ; | number of entries stored here |
| mov | edx, 'PAMS' | ; | 'SMAP' |
| mov | eax, 0xe820 |  |  |
| mov | ecx, 24 | ; | memory map entry struct is 24 bytes |
| int | 0x15 | ; | get first entry |
| jc | .error |  |  |
| cmp | eax, 'PAMS' | ; | bios returns SMAP in eax |
| jne | .error |  |  |
| test | ebx, ebx | ; | if ebx=0 then list is one entry long; bail out |
| je | .error |  |  |
| jmp | .start |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| mov | edx, 'PAMS' | ; | some bios's trash this register |
| mov | ecx, 24 | ; | entry is 24 bytes |
| mov | eax, 0xe820 |  |  |
| int | 0x15 | ; | get next entry |
| jcxz | .skip\_entry | ; | if actual returned bytes is 0, skip entry |

|  |  |  |  |
| --- | --- | --- | --- |
| inc | bp | ; increment entry count | |
| add | di, 24 | | ; point di to next entry in buffer | |

|  |  |  |
| --- | --- | --- |
| cmp | ebx, 0 | ; if ebx return is 0, list is done |
| jne | .next\_entry | ; get next entry |
| jmp | .done |  |

stc

.done:

popad ret

### 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が、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い きます。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ます。

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に行われています。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。

.vbe\_interface\_seg resw 1

.vbe\_interface\_off resw 1

.vbe\_interface\_len resw 1 endstruc

resw

.vbe\_mode

; required

; memory size. Present if flags[0] is set

; boot device. Present if flags[1] is set

; kernel command line. Present if flags[2] is set

; number of modules loaded along with kernel. present if flags[3] is set

; symbol table info. present if flags[4] or flags[5] is set

; memory map. Present if flags[6] is set

; phys address of first drive structure. present if flags[7] is set

; ROM configuation table. present if flags[8] is set

; Bootloader name. present if flags[9] is set

; advanced power management (apm) table. present if flags[10] is set

; video bios extension (vbe). present if flags[11] is set

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

1

.vbe\_control\_info resd

.vbe\_mode\_info resd

struc multiboot\_info

.flags resd

.memoryLo resd

.memoryHi resd

.bootDevice resd

.cmdLine resd

.mods\_count resd

.mods\_addr resd

.syms0 resd

.syms1 resd

.syms2 resd

.mmap\_length resd

.mmap\_addr resd

.drives\_length resd

.drives\_addr resd

.config\_table resd

.bootloader\_name resd

.apm\_table resd

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。

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 mov mov | eax, 0x2BADB002  ebx, 0  edx, [ImageSize] | ; multiboot specs say eax should be this |
| push call add | dword boot\_info ebp  esp, 4 | ; Execute Kernel |
| cli hlt |  |  |

...そして、我々のカーネルの中にも。

//! kernel entry point is called by boot loader

void cdecl kernel\_entry (multiboot\_info\* bootinfo) {

//\*snip\*

}

カーネルの multiboot\_info 構造体は、上に示したものと同じですが、C 言語で書かれています。この設定のおかげで、カーネルが行うべきことは、

bootinfo を通してメモリ情報 (およびブートローダから渡される情報) 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っ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することが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、各ブロックを把握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つのビットで表すのです。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。

PMMNGR\_BLOCK\_SIZE

//! block alignment #define PMMNGR\_BLOCK\_ALIGN

4096

//! block size (4k) #define PMMNGR\_BLOCK\_SIZE

//! 8 blocks per byte

#define PMMNGR\_BLOCKS\_PER\_BYTE 8

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。

//! size of physical memory

static uint32\_t \_mmngr\_memory\_size=0;

//! number of blocks currently in use

static uint32\_t \_mmngr\_used\_blocks=0;

//! maximum number of available memory blocks static uint32\_t \_mmngr\_max\_blocks=0;

//! memory map bit array. Each bit represents a memory block static uint32\_t\* \_mmngr\_memory\_map= 0;

その中でも最も重要なのが\_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含みます。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で表したものです。

メモリのビットマップ

じゃあ、いいですか！？\_mmngr\_memory\_mapは、uint32\_tへのポインタ...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」と考えた方がいいでしょう。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ではありません。

inline void mmap\_set (int bit) {

\_mmngr\_memory\_map[bit / 32] |= (1 << (bit % 32));

}

ビットは0からxまでの値で、xはメモリマップ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。

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。mmp\_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。

inline void mmap\_unset (int bit) {

\_mmngr\_memory\_map[bit / 32] &= ~ (1 << (bit % 32));

}

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。上のルーチンと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) {

return \_mmngr\_memory\_map[bit / 32] & (1 << (bit % 32));

}

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。

int mmap\_first\_free () {

//! find the first free bit

for (uint32\_t i=0; i< pmmngr\_get\_block\_count() / 32; i++) if (\_mmngr\_memory\_map[i] != 0xffffffff)

for (int j=0; j<32; j++) { //! test each bit in the dword

int bit = 1 << j;

if (! (\_mmngr\_memory\_map[i] & bit) ) return i\*4\*8+j;

}

return -1;

}

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、そのドワーフがすべて設定されているかどうかをテストします。この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。 0xffffffでないことを確認してテストします。0xffffff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。

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ブロッ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ようであれば、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ーケータ

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) {

//! By default, all of memory is in use

memset (\_mmngr\_memory\_map, 0xf, pmmngr\_get\_block\_count() / PMMNGR\_BLOCKS\_PER\_BYTE );

}

|  |  |  |
| --- | --- | --- |
| \_mmngr\_memory\_size  \_mmngr\_memory\_map | =  = | memSize;  (uint32\_t\*) bitmap; |
| \_mmngr\_max\_blocks | = | (pmmngr\_get\_memory\_size()\*1024) / PMMNGR\_BLOCK\_SIZE; |
| \_mmngr\_used\_blocks | = | pmmngr\_get\_block\_count(); |

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いたメモリ領域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++);

\_mmngr\_used\_blocks--;

}

mmap\_set (0); //first block is always set. This insures allocs cant be 0

}

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 ションエラーに対して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。

void pmmngr\_deinit\_region (physical\_addr base, size\_t size)

{ int align = base / PMMNGR\_BLOCK\_SIZE; int blocks = size / PMMNGR\_BLOCK\_SIZE;

for (; blocks>0; blocks--)

{ mmap\_set (align++);

\_mmngr\_used\_blocks++;

}

}

わーい。これで、メモリの使用領域を初期化・非初期化する方法と、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・解除に取り組めます。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。

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だけで

す。mmap\_first\_free() ルーチンを使ってビットマップを調べれば、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\_frame () から返されたの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。

}

mmap\_set (frame);

physical\_addr addr = frame \* PMMNGR\_BLOCK\_SIZE;

\_mmngr\_used\_blocks++;

return (void\*)addr;

//out of memory

if (frame == -1)

return 0;

void\* pmmngr\_alloc\_block () {

if (pmmngr\_get\_free\_block\_count() <= 0) return 0; //out of memory

int frame = mmap\_first\_free ();

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。

void pmmngr\_free\_block (void\* p) {

physical\_addr addr = (physical\_addr)p; int frame = addr / PMMNGR\_BLOCK\_SIZE;

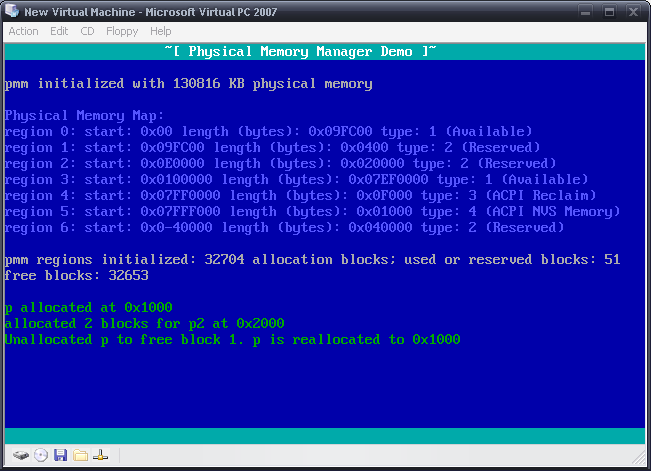
mmap\_unset (frame);

\_mmngr\_used\_blocks--;

}

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。

**Demo**



*VirtualPC上で動作するPhysical Memory Manager*

デモダウンロード(MSVC++)

少しスパイ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う点で、非常に素晴らしいものです。このデモは、すべての光学化レベルで動作するはずです。

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 モリの話を始めるときに、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、物理メモリそのものを見て、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ス指定モードについて理解しました。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、仮想アドレス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。



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メモリマネージャを開発しました。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。

Virtual Memory

Memory Management Unit (MMU)

Translation Lookaside Buffer (TLB)

PAE and PSE

Paging Methods

Pages and Page Faults

The Page Table

The Page Directory Table Implimenting Paging

...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

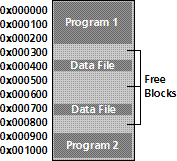
仮想化の必要性

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。



ここで何が起こっ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 トロールする必要が出てきます。そのために必要なのが、物理的なメモリを抽象化する方法です。

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。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 メモリアドレッシングスキームです。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と、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。

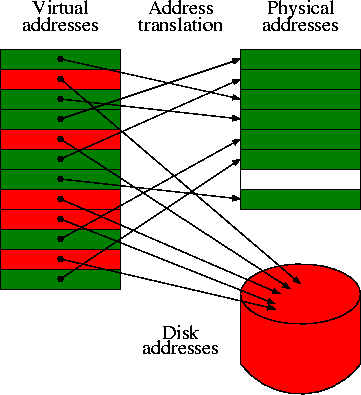
これは重要なことです。ページはページテーブルに格納されています。このページテーブルを設定することで、物理アドレスが仮想アドレスにどのように変換されるかを記述します。言い換えれば TLBは、私たちが設定したページテーブルを使って、仮想アドレスを物理アドレスに変換しています。そう、どの仮想アドレスが何に対応するかを設定するのです。この方法については後ほどご紹介します。心配しなくても、そんなに悪いことではありません。）

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する方法でもあります。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。



ここで何が起こっ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。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、難しいと思われてもご心配なく。）

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ーの機能の一つで、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ライン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して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？

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？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ー（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。

**Bit 0 (P):** Present flag

0: Page is not in memory

1: Page is present (in memory)

**Bit 1 (R/W):** Read/Write flag

0: Page is read only

1: Page is writable

**Bit 2 (U/S):**User mode/Supervisor mode flag

0: Page is kernel (supervisor) mode

1: Page is user mode. Cannot read or write supervisor pages

**Bits 3-4 (RSVD):** Reserved by Intel

**Bit 5 (A):** Access flag. Set by processor

0: Page has not been accessed

1: Page has been accessed

**Bit 6 (D):** Dirty flag. Set by processor

0: Page has not been written to

1: Page has been written to

**Bits 7-8 (RSVD):** Reserved

**Bits 9-11 (AVAIL):** Available for use

**Bits 12-31 (FRAME):** Frame address

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。

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

ebx, 0x100000 | PRIV ; this page is mapped to 1MB

mov

3

PRIV

%define

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」（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、これは真実です）、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, | //0000000000000000000000000000001 |
| I86\_PTE\_WRITABLE | = | 2, | //0000000000000000000000000000010 |
| I86\_PTE\_USER | = | 4, | //0000000000000000000000000000100 |
| I86\_PTE\_WRITETHOUGH | = | 8, | //0000000000000000000000000001000 |
| I86\_PTE\_NOT\_CACHEABLE | = | 0x10, | //0000000000000000000000000010000 |
| I86\_PTE\_ACCESSED | = | 0x20, | //0000000000000000000000000100000 |
| I86\_PTE\_DIRTY | = | 0x40, | //0000000000000000000000001000000 |
| I86\_PTE\_PAT | = | 0x80, | //0000000000000000000000010000000 |
| I86\_PTE\_CPU\_GLOBAL | = | 0x100, | //0000000000000000000000100000000 |
| I86\_PTE\_LV4\_GLOBAL | = | 0x200, | //0000000000000000000001000000000 |
| I86\_PTE\_FRAME | = | 0x7FFFF000 | //1111111111111111111000000000000 |
| }; |  |  |  |

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イ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。

//! page table entry typedef uint32\_t pt\_entry;

簡単ですね。次は、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);

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

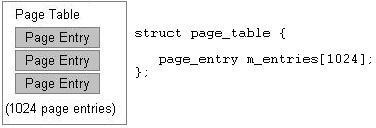
ッピングされているかを追跡することができます。このテーブルの各ページエントリは、前のセクションで示したフォーマットに従 います。つまり、ページテーブルとは、ページテーブルエントリ（PTE）の配列のことです。

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。



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。

page table index offset into page

BBBBBBBBBBBB

AAAAAAAAAA

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]

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。

page table index offset into page

000000000000 ; 0xc0000 in binary form BBBBBBBBBBBB

11000000

AAAAAAAAAA

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。ページフォールトハンドラは、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、何だと思いますか？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？

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、ど れだけ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ー）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さ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よく似ています。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。

**Bit 0 (P):** Present flag

0: Page is not in memory

1: Page is present (in memory)

**Bit 1 (R/W):** Read/Write flag

0: Page is read only

1: Page is writable

**Bit 2 (U/S):**User mode/Supervisor mode flag

0: Page is kernel (supervisor) mode

1: Page is user mode. Cannot read or write supervisor pages

**Bit 3 (PWT):**Write-through flag

0: Write back caching is enabled

1: Write through caching is enabled

**Bit 4 (PCD):**Cache disabled

0: Page table will not be cached

1: Page table will be cached

**Bit 5 (A):** Access flag. Set by processor

0: Page has not been accessed

1: Page has been accessed

**Bit 6 (D):** Reserved by Intel

**Bit 7 (PS):** Page Size

0: 4 KB pages

1: 4 MB pages

**Bit 8 (G):** Global Page (Ignored)

**Bits 9-11 (AVAIL):** Available for use

**Bits 12-31 (FRAME):** Page Table Base address

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ー（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 and pde.cpp - Abstracting Page Directory Entries**

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 {

};

//! a page directery entry typedef uint32\_t pd\_entry;

|  |  |  |  |
| --- | --- | --- | --- |
| I86\_PDE\_PRESENT | = | 1, | //0000000000000000000000000000001 |
| I86\_PDE\_WRITABLE | = | 2, | //0000000000000000000000000000010 |
| I86\_PDE\_USER | = | 4, | //0000000000000000000000000000100 |
| I86\_PDE\_PWT | = | 8, | //0000000000000000000000000001000 |
| I86\_PDE\_PCD | = | 0x10, | //0000000000000000000000000010000 |
| I86\_PDE\_ACCESSED | = | 0x20, | //0000000000000000000000000100000 |
| I86\_PDE\_DIRTY | = | 0x40, | //0000000000000000000000001000000 |
| I86\_PDE\_4MB | = | 0x80, | //0000000000000000000000010000000 |
| I86\_PDE\_CPU\_GLOBAL | = | 0x100, | //0000000000000000000000100000000 |
| I86\_PDE\_LV4\_GLOBAL | = | 0x200, | //0000000000000000000001000000000 |
| I86\_PDE\_FRAME | = | 0x7FFFF000 | //1111111111111111111000000000000 |

難しいことではありません。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);

extern bool pd\_entry\_is\_user (pd\_entry);

extern bool pd\_entry\_is\_4mb (pd\_entry);

extern bool pd\_entry\_is\_writable (pd\_entry e); extern physical\_addr pd\_entry\_pfn (pd\_entry e);

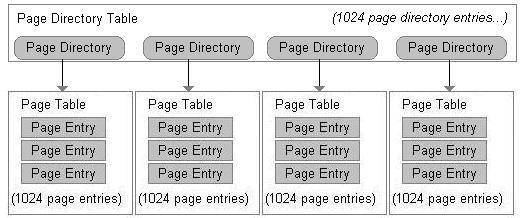
extern void pd\_entry\_enable\_global (pd\_entry e);

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。すごいでしょう？

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 ていますか？

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

ハイヤ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 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いてください。この値が0xffffffよりも大きくなると、折り返して0に戻ります。0xffffff - 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レクタが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 にベースセレ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のセレクタのベース（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+ベースセレクタ((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。

|  |  |  |
| --- | --- | --- |
| AAAAAAAAAA | BBBBBBBBBB | CCCCCCCCCCCC |
| directory index | page table index | offset into page |

これは非常に重要です。これはプロセッサー（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れます。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が 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

てくれます。

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。何のトリックもありません。

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。

|  |  |  |  |
| --- | --- | --- | --- |
| 1100000000  AAAAAAAAAA | 0000000000  BBBBBBBBBB | 000000000000  CCCCCCCCCCCC | ; 0xC0000000 in binary form |
| directory index | page table index | offset into page |  |

ディレクトリインデックスは、ページディレクトリテーブル内のどのページテーブルにアクセスしているかを教えてくれるこ

とを覚えていますか？つまり...1100000000b（ディレクトリインデックス）＝768番目のページテーブルということになります。

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。

あとは、768番目のページテーブルの最初のページのフレームアドレスを0x100000に設定すれば、出来上がりです。これで3GBの仮想アドレスが1MBの物理アドレスにマッピングされたことになります。各ページのアラインメントが4KBであることを考えると、この作業を4KB の物理アドレス単位で続けることができます。

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

これですべてだと思います。次に見ていくのは、このチュートリアルのために開発された仮想メモリマネージャ（VMM）そのものです。こ れまで見てきたものをまとめて、すべてがどのように動作するかを確認します。

私は、まだ見なければならない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。

//! virtual address

typedef uint32\_t virtual\_addr;

//! i86 architecture defines 1024 entries per table--do not change #define PAGES\_PER\_TABLE 1024

#define PAGES\_PER\_DIR 1024

#define PAGE\_DIRECTORY\_INDEX(x) (((x) >> 22) & 0x3ff)

#define PAGE\_TABLE\_INDEX(x) (((x) >> 12) & 0x3ff) #define PAGE\_GET\_PHYSICAL\_ADDRESS(x) (\*x & ~0xfff)

//! page table represents 4mb address space #define PTABLE\_ADDR\_SPACE\_SIZE 0x400000

//! directory table represents 4gb address space #define DTABLE\_ADDR\_SPACE\_SIZE 0x100000000

//! page sizes are 4k #define PAGE\_SIZE 4096

//! page table struct ptable {

pt\_entry m\_entries[PAGES\_PER\_TABLE];

};

//! page directory struct pdirectory {

pd\_entry m\_entries[PAGES\_PER\_DIR];

};

physical\_addrタイプと同様に、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 ジテーブルエントリの配列に過ぎない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。）

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フォーマット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。

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。DTABLE\_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。

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、4Kページのみを管理します。

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。

bool vmmngr\_alloc\_page (pt\_entry\* e) {

//! allocate a free physical frame void\* p = pmmngr\_alloc\_block (); if (!p)

return false;

//! map it to the page pt\_entry\_set\_frame (e, (physical\_addr)p);

pt\_entry\_add\_attrib (e, I86\_PTE\_PRESENT);

return true;

}

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、使用できるようになります。いいでしょう？

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 定するということです。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

ページを解放するのはもっと簡単です。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ーPRESENT ビットをクリア（NOT PRESENTと表示）するだけです。

void vmmngr\_free\_page (pt\_entry\* e) {

void\* p = (void\*)pt\_entry\_pfn (\*e); if (p)

pmmngr\_free\_block (p);

pt\_entry\_del\_attrib (e, I86\_PTE\_PRESENT);

}

これで完成です。さて、1つのページ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...。

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

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、エントリー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) ]; return 0;

}

vmmngr\_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（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。vmmngr\_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。

//! current directory table (global) pdirectory\* \_cur\_directory=0;

inline bool vmmngr\_switch\_pdirectory (pdirectory\* dir)

{ if (!dir)

return false;

\_cur\_directory = dir; pmmngr\_load\_PDBR (\_cur\_pdbr); return true;

}

pdirectory\* vmmngr\_get\_directory ()

{ return \_cur\_directory;

}

vmmngr\_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すか？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。

void vmmngr\_flush\_tlb\_entry (virtual\_addr addr)

{ #ifdef \_MSC\_VER

\_asm {

cli

invlpg addr sti

}

#endif

}

INVLPGは特権的な命令であることを覚えておいてください。この命令を使用するには、スーパバイザモードで動作している必要があります。

vmmngr\_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。

void vmmngr\_map\_page (void\* phys, void\* virt) {

//! get page directory

pdirectory\* pageDirectory = vmmngr\_get\_directory ();

//! get page table

pd\_entry\* e = &pageDirectory->m\_entries [PAGE\_DIRECTORY\_INDEX ((uint32\_t) virt) ]; if ( (\*e & I86\_PTE\_PRESENT) != I86\_PTE\_PRESENT) {

パラメーターとして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インデックスは仮想アドレス自体の一部なので、PAGE\_DIRECTORY\_INDEX()を使ってページディレクトリインデック スを取得します。そして、ページディレクトリの配列にインデックスして、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ビットが設定されているかどうかをテストします。セットされていなければ、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...。

//! page table not present, allocate it

ptable\* table = (ptable\*) pmmngr\_alloc\_block (); if (!table)

return;

//! clear page table

memset (table, 0, sizeof(ptable));

//! create a new entry pd\_entry\* entry =

&pageDirectory->m\_entries [PAGE\_DIRECTORY\_INDEX ( (uint32\_t) virt) ];

//! map in the table (Can also just do \*entry |= 3) to enable these bits pd\_entry\_add\_attrib (entry, I86\_PDE\_PRESENT);

pd\_entry\_add\_attrib (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。

//! get table

ptable\* table = (ptable\*) PAGE\_GET\_PHYSICAL\_ADDRESS ( e );

//! get page

pt\_entry\* page = &table->m\_entries [ PAGE\_TABLE\_INDEX ( (uint32\_t) virt) ];

//! map it in (Can also do (\*page |= 3 to enable..) pt\_entry\_set\_frame ( page, (physical\_addr) phys); pt\_entry\_add\_attrib ( page, I86\_PTE\_PRESENT);

}

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さ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。

void vmmngr\_initialize () {

//! allocate default page table

ptable\* table = (ptable\*) pmmngr\_alloc\_block (); if (!table)

return;

//! allocates 3gb page table

ptable\* table2 = (ptable\*) pmmngr\_alloc\_block (); if (!table2)

return;

//! clear page table vmmngr\_ptable\_clear (table);

ページテーブルが4Kアラインド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すか？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、ページテーブルはそのエントリに対して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（ページテーブルエントリのサイズ） = 4K）ので、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付けて、私たちが使うためにきれいにします。

//! 1st 4mb are idenitity mapped

for (int i=0, frame=0x0, virt=0x00000000; i<1024; i++, frame+=4096, virt+=4096) {

//! create a new page

pt\_entry page=0;

pt\_entry\_add\_attrib (&page, I86\_PTE\_PRESENT); pt\_entry\_set\_frame (&page, frame);

//! ...and add it to the page table

table2->m\_entries [PAGE\_TABLE\_INDEX (virt) ] = page;

}

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IDENITITYマッピングです。

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。

//! map 1mb to 3gb (where we are at)

for (int i=0, frame=0x100000, virt=0xc0000000; i<1024; i++, frame+=4096, virt+=4096) {

//! create a new page pt\_entry page=0;

pt\_entry\_add\_attrib (&page, I86\_PTE\_PRESENT); pt\_entry\_set\_frame (&page, frame);

//! ...and add it to the page table

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。

//! create default directory table

pdirectory\* dir = (pdirectory\*) pmmngr\_alloc\_blocks (3); if (!dir)

return;

//! clear directory table and set it as current 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。

pd\_entry\* entry = &dir->m\_entries [PAGE\_DIRECTORY\_INDEX (0xc0000000) ]; pd\_entry\_add\_attrib (entry, I86\_PDE\_PRESENT);

pd\_entry\_add\_attrib (entry, I86\_PDE\_WRITABLE); pd\_entry\_set\_frame (entry, (physical\_addr)table);

pd\_entry\* entry2 = &dir->m\_entries [PAGE\_DIRECTORY\_INDEX (0x00000000) ]; pd\_entry\_add\_attrib (entry2, I86\_PDE\_PRESENT);

pd\_entry\_add\_attrib (entry2, I86\_PDE\_WRITABLE); pd\_entry\_set\_frame (entry2, (physical\_addr)table2);

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いますか？各ページディレクトリ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れば、各ページディレクトリエントリがディレクトリテーブル全体の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言っても差し支えないでしょう。ページディレクトリの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。

同様にして、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。

//! store current PDBR

\_cur\_pdbr = (physical\_addr) &dir->m\_entries;

//! switch to our page directory vmmngr\_switch\_pdirectory (dir);

//! enable paging

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に要求する場合が多々あります。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: #PF occured because page was present

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 0: #PFの原因となった操作が読み取りだった場合 1:

す。

#PFの原因となった操作が書き込みだった場合

Bit 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

Bit 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:

Bit 4: 予約済みのビットが書き込まれたため、#PFが発生した

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、ページフォー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。

**Demo**

このデモには、このチュートリアルの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ー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N eptune Operating Systemを開発中 質問やコメントは？お気軽にご連絡ください。*

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

ようこそ!

この章では、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ー、キーボードドライバーの開発などを紹介します。

キーボードは、このシリーズで初めてプログラムするデバイスでもあります。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、経験を積んできたので、今度はそれを試す番です。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、そして我々のシステムと通信します。さらに複雑なことに、2つのコントローラーには、それぞれ独自のコマンドと操作方法があります。そのため、本章ではいくつかの箇所でかなり詳細に説明しています。

また、この章では、最初のインタラクティブなデモを紹介します。基本的なコマンドラインパーサーです。楽しみですか？

本章は、デバイスドライバをより深く掘り下げて説明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。

Heres the list:

Keyboard - Back in Time and Keyboard Layout Inside the keyboard

Keyboard Protocols Keyboard Encoder Keyboard Controller Scan Code Sets Keyboard IRQ

Lets get going!

**Keyboard - History**

**Back in time**

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、キーボードの誕生はタイプライターをそのままモデルにしたわけではない。

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した際、複数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ました。その結果、生まれたのが「テレグラフ」です。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取り入れていた。

ENIAC（Electronic Numerical Integrator And Computer）は、最初の汎用コンピューターである。ENIACは、パンチカードリーダーを入力装置と出力装置の両方として使用した。

1948年、BINAC（BINary Automatic Computer）は、入力と出力の両方に電気機械的に制御されたタイプライターを使用した。

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？現在のようなキーボードに進化したのは、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。テレビに使われているブラウン管（CRT）の技術と、パソコンに使われているコンピュータの技術を組み合わせたのだ。