OS が動作する x86 エミュレータの開発

坂本優太

1　背景

[1, 30 日でできる！ OS 自作入門] という本をきっかけにプログラミングを初めた僕は，2016 年のセキュリティ・キャンプ全国大会に参加し，よりコンピュータの仕組みに興味を持つようになった．その後，僕は [2, 自作エミュレータで学ぶ x86 アーキテクチャ] を読んだことで，[1] では自作 OS の動作確認のために使っているだけだったエミュレータというプログラムの仕組みにも興味を持つようになった．

エミュレータとは，コンピュータの動作をエミュレート，つまり模倣するプログラムのことで，代表的なものとしては QEMU[*1] や Bochs[*2] などがある．これらのエミュレータはコンピュータをほぼ正確に模倣するため，Windows や Linux などの主要な OS や，「はりぼて OS」[*3] を動かすことができる．このように，仮想的なコンピュータ上でソフトウェアを動作させることを仮想化といい，これらの技術は最近では OS や別アーキテクチャのソフトウェアの動作確認だけでなく，作業環境の分離などにも用いられる一般的な技術となっている[*4]．

しかし，[2] で作るエミュレータはとても小さく機能もとても少ないものであり，Windows や Linux などの OS を動かすことができないのはもちろん，「はりぼて OS」のような小さな OS も動かすことはできなかった．そこで，これらの OS を動かすために必要なエミュレータの機能を考え，独自のエミュレータを開発を初めた．ただし，Windows や Linux のような大きな OS を動かすためにはとても多くの機能を実装する必要があるため，あくまで「はりぼて OS」を動かすことを目標とした．

2　成果の具体的な内容

2.1　エミュレータの設計

当初は [2] の C で実装されたエミュレータを改良しようとしていたが，多くの機能を実装しようとするとコードが煩雑になってしまう．そこで，改良するのではなく 1 から作り直すことで既に実装されている部分についても十分に理解し，その上で新たな機能の実装も行っていきたいと考えた．そのため，多少は使い慣れていた C++ を使い，C でのコードを参考にしつつも 1 から実装することにした[*5]．

C++ によるエミュレータの実装の基本的な部分はソースコード 1 のようになった．

ソースコード 1　エミュレータの基本的な実装

```
typedef union {
    uint32_t reg32;
    uint16_t reg16; // 下位16bit
    struct {
        uint8_t low8, high8;
    };
} Register32; // 32bit register

class Emulator {
    public:
        Register32 CR[5], eflags, eip, reg[8];  // レジスタ
        uint8_t *memory;                        // メモリ
    private:
        int bite_mode, memory_size;
}

typedef void instruction_func_t(Emulator *emu); // 1つの命令に対応する関数の型
instruction_func_t* instruction16[256];
instruction_func_t* instruction32[256];
```

Emulator クラスは，コンピュータを構成する要素である CPU やメモリを 1 つにまとめたもので，このクラスがエミュレータの本体となる．Register32 は 32bit 長のレジスタを表す共用体で，メンバを使って下位の 8,16bit の領域へのアクセスが行いやすくなっている．これは実際のレジスタでもそのようなアクセスがあるためで，例えば EAX というレジスタは下位 16bit が AX，そのうち上位 8bit が AH，下位 8bit が AL という名前のレジスタとして扱われる．また，メモリは uint8_t 型の並んだメモリ領域として表現した．

こうすると後は，この仮想的なメモリ上に実行するプログラムを置いて，実行するアドレスにある機械語を読み込み (fetch)，意味を解析 (decode) して実行する，という一連の処理を無限ループすればエミュレータになる．ここで，実行するアドレスはプログラ

[*1] Fabrice Bellard が中心となって開発しているオープンソースのエミュレータ．動的バイナリ変換 (Dynamic Binary Translation) などの機能を持ち，高速に動作するのが特徴．
[*2] オープンソースの PC/AT 互換機のエミュレータ．
[*3] [1] で作る小さな OS．
[*4] ただし，これは正確には準仮想化と呼ばれるものの方が多い．
[*5] このときのリポジトリは https://github.com/sk2sat/vm

ム・カウンタという専用のレジスタの値を使うことになっている．x86 では EIP がこれに相当するため，このプログラムでいうと memory[eip.reg32] が実行される機械語の最初のバイトになる．

実行される 1 つ 1 つの機械語命令はそれぞれ 1 つの関数で表現されており，Emulator クラスはこれらの命令のポインタの配列を持っている．こうしておくことで，オペコード n の命令を実行する時は instruction[n] が指す関数を実行すればよく，実装を分かりやすくすることができる．これは [2] の実装を元にしたものだ．しかし，本の実装では命令は 32bit モードのみなので，複数の動作モードに対応するため，16bit と 32bit それぞれに関数ポインタ配列を用意して，動作モードによって使用する配列を分岐するという実装をした．

2.2 ディスプレイの実装

QEMU や Bochs などのエミュレータでは，ディスプレイのエミュレートをすることもできる．エミュレータには必ずしもディスプレイのエミュレートが必要という訳ではない．しかし，動作目標の「はりぼて OS」は比較的 GUI の比重が大きい他，単純に「はりぼて OS」の UI が自分のエミュレータで見られると嬉しいと思ったため，実装することにした．

ディスプレイは，指定した色で発光するピクセルが解像度の分だけ並んでいるデバイスであり，描画はそれぞれの座標への色の指定という形で行われている．この座標と色の設定は，実際には VRAM というメモリ領域への書き込みによって実現されている．

この VRAM では 1 ピクセルに相当するのは 1byte となっている．これでは RGB で色を指定することができないので，ここに書き込むデータは RGB のような色データではなく，パレットという外部装置に事前に設定しておいた色の番号ということになっている．

これらをエミュレートするため，主に以下の 4 つを実装した．

- 外部デバイスへのアクセス
- パレット
- VRAM のデータから RGB データを生成
- RGB データをウィンドウに表示

x86 では，外部デバイスへのアクセスは in/out 命令で行われる．これらのデバイスは 1 つずつ C++ のクラスで表現することとし，デバイスの基本クラスを用意して，それぞれのデバイスの実装はそのクラスを継承したクラスで行うことにした．デバイスの基本クラスでは in/out 用の関数が純粋仮想関数として定義してあり，継承先のデバイスクラスでは必ずこの関数を実装しなければならない．エミュレータは in/out 命令を実行すると，事前に登録されたデバイスクラスの中から指定されたポートに対応するクラスのポインタを選択し，そこから in/out 用の関数を実行する．ディスプレイもこのデバイスクラスとして作り，RGB のテーブルとして表現したパレットを out 命令で設定できるようにした．

VRAM のデータから実際に描画される RGB のデータを生成する部分はソースコード 2 のように，画面上の全ての座標において VRAM から読み取った色番号に対応する色データをパレットから取得して設定することで実装した．

ソースコード 2　パレットのデータを使って描画データを生成

```
unsigned char* Display::Draw(){
    for(int x=0;x<scrnx;x++){
        for(int y=0;y<scrny;y++){
            int i = y*scrnx + x;
            char n= vram[i]; // (x,y)の色番号
            img[i*3  ] = palette[n*3  ]; // Red
            img[i*3+1] = palette[n*3+1]; // Green
            img[i*3+2] = palette[n*3+2]; // Blue
        }
    }
    return img;
}
```

生成した RGB データをウィンドウに表示する部分は，オープンソースでマルチプラットフォーム対応の GUI ライブラリである freeglut を使用した．[*6]

2.3 テスト環境の整備

「はりぼて OS」を動作させることが最終的な目標ではあったが，初めから「はりぼて OS」の最終バージョンを動かそうとしてしまうと，どこまでは動作していて，どこからが動作していないのか，どこにバグが残っているのかという問題を切り分けて考えることがとても難しくなってしまう．そこで，エミュレータのテストを行うためのバイナリについても自分で整備することにした．

[1] では，本文中で「はりぼて OS」の開発がインクリメンタルに進められていく．そのため，「はりぼて OS」は初めの方ほど機能が少ないためエミュレートもしやすく，段々と機能が実装されていくにつれてエミュレータに要求される機能も増えていく．また，「はりぼて OS」は章・節と対応した，”harib01c”というようなバージョン名が付けられている．僕はこれを利用して，特定のバージョンの「はりぼて OS」をビルドするための環境を作り，それを使ってエミュレータのテストをすることで，自分のエミュレータで動作する「はりぼて OS」のバージョンを少しずつ上げていく開発スタイルを確立した．

[*6] 一度作り直した後はより使い勝手のよい GLFW を使用した．

このために作ったのが haribote-os というリポジトリ[*7]で，これを使うと，「はりぼて OS」のバージョンごとに対応したブランチやタグを選択して”make”コマンドを実行するだけで，指定のバージョンの「はりぼて OS」のバイナリを生成することができる．

これにより，エミュレータ[*8]の方のリポジトリで”make run HARIB_VER=harib01c”のようにして「はりぼて OS」のバージョンを指定したテストが行えるようになった．

2.4 セグメンテーションの実装

x86 には，「セグメンテーション」という機能がある．これはメモリ管理機能[*9]の 1 つで，メモリを用途ごとに分割する役割と，アクセス制御を行う役割がある．セグメンテーションでは，メモリ領域をセグメントという単位で分割する．そして，それぞれのセグメントの属性を設定することで，メモリ領域の管理やアクセス制御を行うことができる．

メモリ領域をセグメントに分けることによる最も大きなメリットは，物理アドレスを気にしてプログラミングする必要がなくなる，という点にある．例えば，あるセグメントの開始アドレスが物理アドレスで 0x10000 とする．このとき，このセグメントをデータ用のセグメントと指定した上で，0x100 というアドレスにあるデータを取得する命令を実行すると，CPU のアドレス変換という機能[*10]により，実際には物理アドレスで 0x10000+0x100=0x10100 にあるデータが取得される．この時の 0x100，つまり変換前のアドレスのことを論理アドレスという．

このようなアドレス変換の実装は，ソースコード 3 のようになった．

ソースコード 3　論理アドレスの物理アドレスへの変換

```
uint32_t Emulator::L2P(const x86::SRegister *sreg, const uint32_t &addr){
    if(!IsProtected())
        return (sreg->reg16 * 16) + addr; // real mode
    Descriptor desc;
    desc.low32 = GET_MEM32(GDTR.base+(sreg->index*8));
    desc.high32= GET_MEM32(GDTR.base+(sreg->index*8)+4);
    return addr + desc.GetBase();
}
```

この関数は，セグメントレジスタと論理アドレスを引数にとり，物理アドレスを返す．セグメントレジスタというのは，どのセグメントを使うのかを指定するためのレジスタで，その用途によって，コードセグメント用の CS，データセグメント用の DS などのものがある．

リアルモードにおいてはアドレス変換はとても単純で，物理アドレスはセグメントレジスタの値に 16 をかけたものに論理アドレスを加算したものになる．

一方，プロテクトモードではセグメントの開始アドレス，サイズ，属性を自由に設定できるようになっている．この属性というのは，データ用なのか，実行可能なのか，実行可能なら 16,32bit のどちらのモードで実行するのか，どの特権レベルからアクセスできるのか，といったものを設定できる．このような保護のための設定ができるのがプロテクトモードが「プロテクト」モードたる所以でもある．

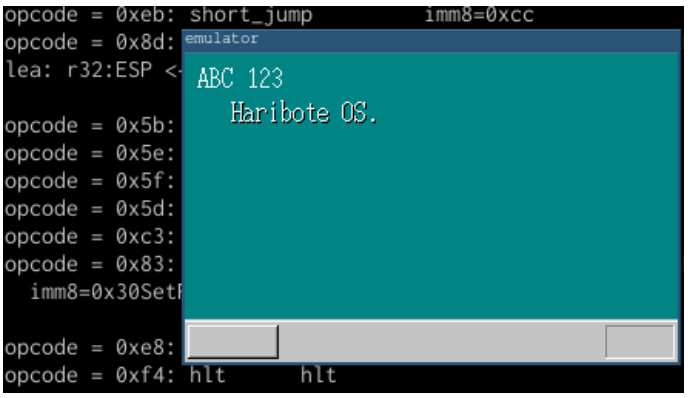
では，このようなセグメントの情報はどのように設定するのかというと，とてもレジスタのような小さな記憶領域には保存できないため，GDT と呼ばれるメモリ領域に保存する．GDT は Global Descriptor Table の略で，1 つのセグメントの属性などをまとめた Descriptor という構造体を連続して並べた領域だ．そして，この GDT 内での Descriptor の番号をセグメントレジスタに入れることで，セグメントを指定することができる．

このため，Descriptor を表す構造体を作り，GDT の開始アドレスにセグメントレジスタの値の分を足したアドレスからデータを取得して Descriptor を取得．その Descriptor からセグメントの開始アドレスを取得して論理アドレスに足すして物理アドレスを計算するという実装になった．

上記のようにして，セグメンテーションのアドレス変換の部分は実装することができた．ただし，アクセス制御を行う部分はまだ CPU の特権レベルなどが存在しないためこれから実装することになる．また，セグメンテーションは簡単な使い方こそ知っていたもののその詳細な動作や仕様はよく分かっていなかったため，実装にあたっては [4, はじめて読む 486] と [3, SDM] が大いに参考になった．

これにより，「はりぼて OS」のブートローダが起動してから，「はりぼて OS」本体をメモリに転送し，プロテクトモードに移行し，セグメントを設定，C 言語で書かれたメイン関数が実行され，起動メッセージが表示される，というところまでを自分のエミュレータで完全に動作させることができるようになった．

図 1　自作エミュレータ上で「はりぼて OS」が起動メッセージを表示するところまで完全に動作するようになった．



[*7] https://github.com/sk2sat/haribote-os

[*8] https://github.com/sk2sat/emu

[*9] メモリ管理機能にはセグメンテーションの他にページングというものもあり，現在ではこちらの方が主流だが，「はりぼて OS」では使用されていないため実装はしなかった．

[*10] 正確には，CPU 内の MMU がこのアドレス変換を行っている．

3　サイボウズ・ラボユースでの活動

僕はこの x86 エミュレータの開発を完全に趣味で行っていた．そんなとき，2017 年の 2 月に東京で開催された OSC [*11]でサイボウズ・ラボユースという学生支援制度を [1] の著者の川合さんに紹介していただいた．これはサイボウズ・ラボが実施しているもので，メンターの指導と奨励金の支給を受けつつ OSS を開発できる制度である．僕は x86 エミュレータの開発をテーマにこの制度に申し込み，書類選考と面接を経て第 7 期サイボウズ・ラボユースにラボユース研究生[*12]として採択された．

ラボユースの期間中，実際に出社できたのは夏休みの間の数回だけのことだったため，開発は基本的に自宅で行い，メンターの光成さんからリモートでコードレビューを頂いたり，絶版となっていた [4] や [5] などの書籍を貸して頂いて参考にしたり，というのがラボユースでの主な活動となった．この他にも，ラボユース内での C++ 勉強会や，同期のラボユース生の方との交流などができ，とても多くの学びが得られた．

2.3 と 2.4 はラボユース期間中の主な成果である．

2018 年 3 月，「第 7 期サイボウズ・ラボユース成果発表会」[*13]で成果発表を行い，僕はサイボウズ・ラボユース研究生を卒業した．

4　今後の方針

現在，「はりぼて OS」の全てのバージョンを動作させることを目標として，新たな x86 エミュレータを 1 から開発している．改良するのではなく作り直す理由は，開発してきたエミュレータの実装は見通しが悪いため，デバッグや改良にかかる労力が大きすぎると判断したからだ．新たな実装は同一リポジトリの v2 ブランチ[*14]で行っている．

v2 の開発では，設計の見通しの良さ[*15]と，デバッグ用のメッセージの分かりやすさをとても重視している．また，実装には C++ の最新の規格である C++17 を用いている．

v2 での最も大きな変更点は，すべての命令の実装に C++ のラムダ式を用いていることだ．これにより，ソースコード 4 のように，x86 の命令を関数型マクロを使ってとても見た目に分かりやすく実装することができるようになった．

ソースコード 4　ラムダ式を使った命令の実装の一部

```
#define INSN(opcode, insn, f, block) \
    name[opcode] = #insn; \
    flag[opcode] = f; \
    func[op] = [](CPU &cpu, std::shared_ptr<Memory> memory) block;

    // 命令の実装
    INSN(0x00, add_rm8_r8, ModRM,{ SET_RM8( ADD(GET_RM8(), GET_R8(REG_NUM)) ); });
    INSN(0x04, add_al_imm8,Imm8, { AL = ADD(AL, IMM8); });
    INSN(0x0c, or_al_imm8, Imm8, { AL = OR( AL, IMM8); });
```

5　単独の成果か否か

これらは全て単独の成果である．実装に当たっては書籍 [2] のプログラムを参考にはしたものの，設計を変更し，機能も大幅に増えたオリジナルのプログラムとなっている．設計・実装はすべて 1 人で行った．ただし，サイボウズ・ラボユース採択期間中は，メンターの光成さんから数回のコードレビューを頂いた．コードレビューで指摘されたのは C++ の書き方に関するものであり，これによる設計やロジックの変更は生じなかった．[*16]

成果は全て GitHub 上で公開している．該当するリポジトリは以下の 3 つである．

- `https://github.com/sk2sat/vm` 当初作っていたエミュレータ
- `https://github.com/sk2sat/emu` ラボユース申し込み後に作り直したエミュレータ
- `https://github.com/sk2sat/haribote-os` 「はりぼて OS」実行テスト用のプロジェクト

参考文献

[1] 30 日でできる！ OS 自作入門
[2] 自作エミュレータで学ぶ x86 アーキテクチャ - コンピュータが動く仕組みを徹底理解！
[3] Intel® 64 and IA-32 Architectures Software Developer’s Manual vol 1,2,3,4
[4] はじめて読む 486 - 32 ビットコンピュータをやさしく語る
[5] Effective C++ - プログラムとデザインを改良するための 55 項目

[*11] Open Source Conference Tokyo/Spring のこと．
[*12] この制度には奨励金の有無のみが異なるラボユース生とラボユース研究生の 2 つのコースがある．僕はラボユース研究生としては初の採択だったようだ．
[*13] `https://blog.cybozu.io/entry/2018/04/05/080000`
[*14] `https://github.com/sk2sat/emu/tree/v2`
[*15] ”数ヶ月後に見直しても難なく読める”ことが目標だ．
[*16] コードレビューによる変更は”[FIX] from code review”という一連のコミットで行われている．