

自作 OS 自動化の PoC としての 遺伝的 MBR

大神祐真 著

2018-10-08 版 へにゃぺんて 発行

はじめに

本書をお手に取っていただきありがとうございます。

本書は自作 OS 自動化の PoC^{*1}として MBR^{*2}作成の自動化を行ってみる本です。

当初は/dev/kvm を直接叩いて VM を作り、自作 OS のテスト自動化のみの予定でしたが、「テストを自動化できるなら、テスト対象である実行バイナリの生成も自動化することで、理論上、自作 OS を自動化できるのでは？」ということで、/dev/kvm を直接叩くところはあまり深く突き詰めず、むしろ手っ取り早く既存のコードを改造してテストツールを作り、テスト結果を遺伝的アルゴリズムで実行バイナリへフィードバックすることで自作 OS 自動化の PoC をしてみることにしました。

MBR にしたのはレガシー BIOS にはなりますが、PC へ電源を入れると直後に動き、なおかつバイナリサイズも 512 バイトと PoC には扱いやすいからです。

本書でやること

自作 OS 自動化には、自作 OS の自動テストツールが必要です。しかし、そのようなものは無さそうなので、既存の比較的行数の少ない VM のサンプルコードを改造して MBR の自動テストツール (MBR テスター) を作ります。既存のものを改造するにしても仮想的な CPU まで全て面倒見るのは大変なので、Linux カーネルが提供する VM 支援機能である KVM を使うサンプルを改造することにします。

ただ、既存のコードを改造するにしても KVM を使った VM がどのようにできているのかを理解していないと既存のコードを理解できません。そのため、改造を始める前に、/dev/kvm を直接叩く簡単なプログラムをいくつか作り、KVM を使う VM がどのようにできているのかを理解します。

そして、自動生成した MBR をテスターで評価し、その結果をフィードバックする仕組み

^{*1} "Proof of Concept"の略。新たなアイデアの実現性を簡単なプログラムで示すこと。

^{*2} "Master Boot Record"の略。UEFI ではないレガシー BIOS が起動時にディスクから読み出して実行してくれるディスク先頭 512 バイトの領域。MBR に含まれる実行バイナリは「ブートストラップローダ」や「イニシャルプログラムローダ (IPL)」と呼ばれる。

みとして「遺伝的アルゴリズム」を使います (遺伝的 MBR)。

そのため、前著風なタイトルとすると、本著は「`/dev/kvm` を直接叩いて KVM の VM を理解し、既存のコードを改造して MBR テスターを作り、遺伝的アルゴリズムで MBR を自動生成する本」です。

本書の構成

本書は 3 章構成で、概要としては以下のとおりです。

- 第 1 章 `/dev/kvm` を直接叩いて理解する
 - 既存のコード改造を始める前に、KVM を使う VM がどのようにできているのかを理解する
 - 「KVM の世界で VM 側に IO を用意して Hello World」と「SeaBIOS を実行」の 2 つを行う
- 第 2 章 既存のコードを改造して MBR テスターを作る
 - KVM を使い SeaBIOS を実行する 5000 行ほどのプログラムである `"kvmulate"` を改造する
 - VM 上の VRAM への文字出力をテストして結果をプロセスの終了ステータスで返すようにする
- 第 3 章 遺伝的 MBR を実現する
 - 512 バイトの MBR 一つ一つを遺伝子として扱い、遺伝的アルゴリズムを適用する
 - 乱数列で生成した MBR を進化させ、何らかの文字を表示させる

なお、予め断っておきますが、第 3 章は「なるほど、そっちに進化してしまったか」というオチです。

開発環境・動作確認環境

筆者が開発や動作確認を行っている環境を以下に記載します。

- PC: Lenovo ThinkPad E450
- OS: Debian GNU/Linux 8.11 (コードネーム: Jessie)
- コンパイラ: GCC 4.9.2
 - `build-essential` パッケージでインストールされるものです

言語としては、1 章と 2 章では C 言語を、3 章ではシェルスクリプトを使う程度なので、一般的な Linux 環境であれば問題ないと思います。2 章で 1 つライブラリのインストールが必要になりますが、パッケージ管理システムで提供されているものなので特に問題はないかと思います (2 章で説明します)。

ただし、KVM を使うため PC としては CPU が仮想化支援機能を持っている必要があります*3。"/dev/kvm" というファイルが存在しない場合、CPU がこれらの機能を持っているか確認してみてください。確認方法は検索すると色々出てくるのでここでは省略します。

また、KVM を使うにはカーネル側で KVM が有効になっている必要もあるのですが、これに関しては比較的最近のディストリビューションであればデフォルトで有効になっているかと思います。これも検索すれば情報が出てくるので必要な場合は調べてみてください。

本書の PDF 版/HTML 版やソースコードの公開場所について

これまでの当サークルの同人誌同様、本書も PDF 版/HTML 版は以下のページで無料で公開しています。

- <http://yuma.ohgami.jp>

扱うソースコードは各章でリポジトリごと異なるので、それぞれの章の中で紹介しています。

*3 Intel であれば Intel VT(vmx)、AMD であれば AMD-V(svm) という名前の機能です。

目次

はじめに	2
本書でやること	2
本書の構成	3
開発環境・動作確認環境	3
本書の PDF 版/HTML 版やソースコードの公開場所について	4
第 1 章 /dev/kvm を直接叩いて理解する	7
1.1 Hello KVM!	7
1.1.1 KVM がやってくれること、KVM で VM を作るためにやらな きゃいけないこと	7
1.1.2 サンプルコード	9
1.1.3 /dev/kvm を open	10
1.1.4 VM 作成	10
1.1.5 ROM 作成	10
1.1.6 CPU 作成	12
1.1.7 CPU レジスタ初期設定	13
1.1.8 実行	14
1.1.9 動作確認	16
1.2 BIOS を動かす	16
1.2.1 サンプルコード	16
1.2.2 この項でやること	17
1.2.3 SeaBIOS を動作させるために必要なこと	18
1.2.4 割り込みコントローラを追加する	18
1.2.5 タイマーを追加する	19
1.2.6 SeaBIOS 自身が入った ROM を追加する	19
1.2.7 RAM を追加する	22

1.2.8	デバッグ用シリアルポートを用意する	23
1.2.9	動作確認	24
第 2 章	既存のコードを改造して MBR テスターを作る	27
2.1	実現すること	27
2.2	手軽そうなサンプルを探す	27
2.3	kvmulate について	28
2.3.1	コンパイル	28
2.3.2	MBR を作成	29
2.3.3	実行するには	30
2.3.4	kvmulate のコード構成	30
2.4	改造する	31
2.5	動作確認	35
2.6	補足: GUI 上の内容を標準出力へ出す	35
第 3 章	遺伝的 MBR を実現する	37
3.1	遺伝的アルゴリズムについて	37
3.2	実装について	38
3.2.1	スクリプト全体の処理の流れ	38
3.2.2	初期個体群生成	42
3.2.3	評価	42
3.2.4	フィードバック	44
3.3	動作確認	49
3.3.1	テストを用意	49
3.3.2	実行結果	49
3.4	補足: 評価方法を工夫する (MBRatoon)	52
おわりに		54
参考情報		55
	参考にさせてもらった情報	55

第 1 章

/dev/kvm を直接叩いて理解する

この章では、`/dev/kvm` を直接叩く簡単なプログラムをいくつか作って、KVM を使う VM がどのように作られているのかを理解します。

ただし、次章で既存のプログラムを改造する際、どこを改造するかは説明するので、特にここらへんの理解が不要な場合、この章を飛ばしても大丈夫です。

1.1 Hello KVM!

Linux カーネルには Kernel-based Virtual Machine(KVM) という機能があり、その名の通り仮想マシンを実現するための機能を提供しています。

KVM は `/dev/kvm` というデバイスファイルでカーネルから提供されています。

この項では `/dev/kvm` を直接叩いて VM を作成し、"Hello KVM!" と表示させてみます。

1.1.1 KVM がやってくれること、KVM で VM を作るためにやらなきゃいけないこと

KVM で VM を作る上で、まずは KVM が何をしてきて、KVM で VM を作るためには何をしなければならないかを説明します。

KVM は VM(構造のみ) と仮想的な CPU(VCPU) といくつかの周辺 IC を用意してくれます。

KVM へ「VM を作って」とリクエストすると KVM は VM を作ってくれます。ただし KVM が作ってくれるのは KVM が VM を管理するための構造のみです。VM 内の各種の仮想的なハードウェアは CPU と一部の周辺 IC を除き KVM を使うアプリ側で用意します。(そして用意した仮想的なハードウェアを KVM が管理する VM へ登録します)

以上から、KVM で VM を作る大まかな流れは以下の通りです。

1. KVM へ VM と VCPU の作成をリクエストする
2. メモリなどを用意し VM へ登録

VM 作成後、VM 実行を KVM へリクエストすると、カーネル側で VM の実行が始まります。

特権が必要な命令が実行された場合や、IO の処理が実行された場合に処理がカーネルから戻ってきます。

そのため、IO などは VM 実行後、カーネルから処理が戻ってきた際にハンドリングします。

この記事で作る VM を図示すると図 1.1 の通りです。

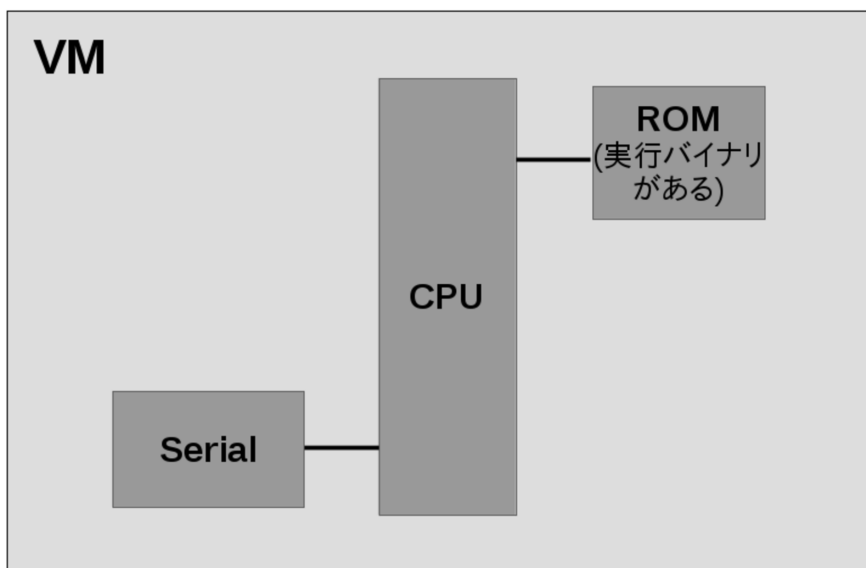


図 1.1 KVM 上で動く VM のアーキテクチャ

ROM("Hello KVM!"とシリアル IO へ出力するプログラムが書かれている) とシリアルが CPU につながっている構成です。

メモリマップは図 1.2 の通りです。

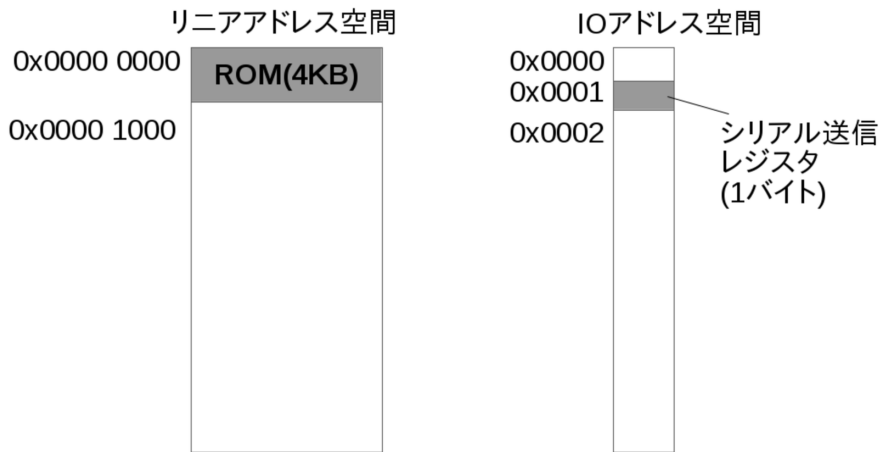


図 1.2 "Hello KVM!"サンプルのメモリマップ

ROM はリニアアドレス空間の 0 番地にマップしていて、シリアル送信のレジスタは IO アドレス空間の 0x01 番地にあることとします。(簡単のためにこうしているだけで、実機や QEMU でこの様になっているわけではありません。)

1.1.2 サンプルコード

この章のサンプルコードは以下のリポジトリで公開しています。

- https://github.com/cupnes/bare_metal_kvm

この項で説明するサンプルは「01_hello」というディレクトリに格納しています。01_hello にはリスト 1.1 のようにファイルが格納されています。

リスト 1.1 01_hello の内容

```
01_hello/
  main.c
  Makefile
  rom/
    Makefile
    rom.ld
    rom.s
```

VM のソースコード自体は main.c のみです。

VM 上で実行する "Hello KVM!" を出力するプログラムは rom ディレクトリに格納しています。rom ディレクトリ内の Makefile でコンパイルすると、バイナリを 16 進の配列に

したヘッダファイル rom.h を生成します (後述)。

01_hello 直下の Makefile に、rom ディレクトリ内の Makefile の呼び出し含め、ビルドルールが記載されています。01_hello 直下で "make" を実行するだけでビルドでき、"vm" という実行バイナリを生成します。"vm" バイナリに上述の rom バイナリも含んでいるため、単体で実行できます。

また他にも、"make run" で実行 (単に "vm" バイナリを実行するだけです)、"make clean" でビルド時の生成物の削除が行えます。

それでは、次節から main.c についてコードブロック毎に紹介してみます。

1.1.3 /dev/kvm を open

まず、/dev/kvm を open し、ファイルディスクリプタを取得します (リスト 1.2)。

リスト 1.2 01_hello/main.c

```
/* /dev/kvm を open */
int kvmfd = open("/dev/kvm", O_RDWR);
}
```

1.1.4 VM 作成

次に、変数 kvmfd へ格納したファイルディスクリプタを使用して KVM へ VM 作成をリクエストします。

リクエストは ioctl を使用して行います (リスト 1.3)。

リスト 1.3 01_hello/main.c

```
/* VM 作成 */
int vmfd = ioctl(kvmfd, KVM_CREATE_VM, 0);
```

以降は、ここで取得した VM のファイルディスクリプタを使用して ioctl でハードウェアを VM へ登録していきます。

1.1.5 ROM 作成

先程取得した vmfd へ ROM を登録します。

まず、今回作る VM 上で "Hello KVM!" を出力する実行バイナリを用意します。

そして、用意した実行バイナリを ROM として vmfd へ登録します。

"Hello KVM!"の実行バイナリについて

rom ディレクトリの Makefile をみると分かりますが、"Hello KVM!"の実行バイナリは rom.h という C 言語のヘッダファイルとして生成されます。(中に unsigned char 配列として実行バイナリが用意されます。)

rom.h は、同じ階層にある rom.s をアセンブルし、rom.ld に従ってリンクした結果の実行バイナリを、xxd コマンドで C の配列の形式へ変換したものです。

そして、rom.s はシリアル送信レジスタの IO アドレス 0x01 へ 1 文字ずつ書き込むアセンブラのプログラムです (リスト 1.4)。

リスト 1.4 01_hello/rom/rom.s

```

mov    $'H',    %al
out    %al,     $0x01
mov    $'e',    %al
out    %al,     $0x01
mov    $'l',    %al
out    %al,     $0x01
mov    $'l',    %al
out    %al,     $0x01
mov    $'o',    %al
out    %al,     $0x01
mov    $' ',    %al
out    %al,     $0x01
mov    $'K',    %al
out    %al,     $0x01
mov    $'V',    %al
out    %al,     $0x01
mov    $'M',    %al
out    %al,     $0x01
mov    $'!',    %al
out    %al,     $0x01
hlt

```

"Hello KVM!"と 1 文字ずつ出力しているのがなんとなくわかるかと思います。mov 命令で 1 文字ずつ"AL"という 1 バイトのレジスタに格納し、out 命令でそれを IO アドレス 0x01 番へ書き込んでいます。

一通り文字を出力した後は hlt 命令を実行するようにしています。今回の VM は「halt 命令で VM 自体終了する」こととしてみます。

そして、実行バイナリを C の配列の形式へ変換して得られる rom.h はリスト 1.5 の通りです。

リスト 1.5 01_hello/rom/rom.h

```
unsigned char rom_bin[] = {
    0xb0, 0x48, 0xe6, 0x01, 0xb0, 0x65, 0xe6, 0x01, 0xb0, 0x6c, 0xe6, 0x01,
    0xb0, 0x6c, 0xe6, 0x01, 0xb0, 0x6f, 0xe6, 0x01, 0xb0, 0x20, 0xe6, 0x01,
    0xb0, 0x4b, 0xe6, 0x01, 0xb0, 0x56, 0xe6, 0x01, 0xb0, 0x4d, 0xe6, 0x01,
    0xb0, 0x21, 0xe6, 0x01, 0xf4
};
unsigned int rom_bin_len = 41;
```

VM 側ではこの rom.h を include し、rom_bin 配列の内容を ROM へ配置するバイナリとして使用します。

ROM を VM へ登録

用意した実行バイナリを ROM として VM へ登録します (リスト 1.6)。

リスト 1.6 01_hello/main.c

```
/* ROM を用意 */
unsigned char *mem = mmap(NULL, ROM_SIZE, PROT_READ|PROT_WRITE,
                          MAP_SHARED|MAP_ANONYMOUS|MAP_NORESERVE,
                          -1, 0);
memcpy(mem, rom_bin, sizeof(rom_bin)); /* メモリヘコードを配置 */
struct kvm_userspace_memory_region region = {
    .guest_phys_addr = 0,
    .memory_size = ROM_SIZE,
    .userspace_addr = (unsigned long long)mem
};
ioctl(vmfd, KVM_SET_USER_MEMORY_REGION, &region); /* VM へメモリを設定 */
```

リスト 1.6 では、mmap で確保した領域へ rom_bin をコピーしています。

その後リスト 1.6 では、kvm_userspace_memory_region 構造体の変数を定義し、KVM_SET_USER_MEMORY_REGION という ioctl のリクエストで mmap で確保した領域を VM へ登録します。

kvm_userspace_memory_region のメンバは最低限必要なもののみ値を設定しています。

guest_phys_addr が VM 上のゲストが見るアドレスで、memory_size がメモリ領域のサイズ (バイト単位)、userspace_addr が VM が確保した領域のアドレスです。

今回の場合、定数 ROM_SIZE(4KB) の大きさの mmap で確保した領域 (先頭アドレスがポインタ変数 mem に入っている) が、VM 上のゲストからは 0x00000000 のアドレスから見えるようになります。

1.1.6 CPU 作成

次に CPU を作成します (リスト 1.7)。

リスト 1.7 01_hello/main.c

```

/* VCPU 作成 */
int vcpufd = ioctl(vmfd, KVM_CREATE_VCPU, 0);
/* 第3引数は作成する vcpu id */
size_t mmap_size = ioctl(kvmfd, KVM_GET_VCPU_MMAP_SIZE, NULL);
struct kvm_run *run = mmap(NULL, mmap_size, PROT_READ|PROT_WRITE,
                           MAP_SHARED, vcpufd, 0);

```

CPU(Virtual CPU、VCPU)の実体はKVM側(カーネル側)にあり、作成する際はKVM_CREATE_VCPUというioctlをリクエストをするだけです。

VCPU作成時にカーネル側で作られるkvm_run構造体へは、後々、VM側からもアクセスします。そのため、KVM_CREATE_VCPUの後、kvm_run構造体へアクセスできるように用意しています。

やっていることは、「カーネル側のkvm_run構造体の領域をVM自身のメモリ空間へマップ(mmap)」です。

VCPUをmmapする際のサイズを取得(KVM_GET_VCPU_MMAP_SIZE)し、mmapでVM自身のメモリ空間にマップします。

mmapの結果はrunというkvm_run構造体のポインタ変数に格納しています。

1.1.7 CPUレジスタ初期設定

KVMのVCPUが持つレジスタの初期値を設定します。

VCPUのレジスタへのアクセスもioctlで行います。KVM_SET_SREGSとKVM_SET_REGSという2つのioctlに分かれており、それぞれで扱えるレジスタが異なります。これらはそれぞれ値を設定するioctlで、逆に取得はKVM_GET_SREGSとKVM_GET_REGSで行えます。

また、これらのioctlへは引数として、KVM_SET_SREGSの場合はkvm_sregs構造体、KVM_SET_REGSの場合はkvm_regs構造体を渡します。これらの構造体を通してレジスタ値の設定/取得を行います。

KVM_SET_SREGS

まずは、KVM_SET_SREGSを実施します(リスト1.8)。

リスト 1.8 01_hello/main.c

```
struct kvm_sregs sregs; /* セグメントレジスタ初期値設定 */
ioctl(vcpufd, KVM_GET_SREGS, &sregs);
sregs.cs.base = 0;
sregs.cs.selector = 0;
ioctl(vcpufd, KVM_SET_SREGS, &sregs);
```

リスト 1.8 では、CPU が実行する命令のアドレスに関する設定を行っています。

ここで、この項の VM は「起動されると 0x00000000 の命令から実行を始める」こととします。そのための「セグメンテーション」と呼ばれる x86 CPU の機能の設定を行っているのがリスト 1.8 です。セグメンテーションとはアドレス空間を「セグメント」と呼ぶ領域に分けてアクセスする方式です。セグメントには用途が決まっているものもあり、リスト 1.8 では「コードセグメント (CS)」という「CPU が実行する命令が配置されているセグメント」の設定を行っています。

セグメンテーションについてここでこれ以上説明はしませんが、やっていることは単に CS がアドレス 0x00000000 から始まる事を設定しているだけです。

KVM_SET_REGS

次に KVM_SET_REGS を実施します (リスト 1.9)。

リスト 1.9 01_hello/main.c

```
struct kvm_regs regs = { /* ステータスレジスタ初期値設定 */
    .rip = 0x0,
    .rflags = 0x02, /* RFLAGS 初期状態 */
};
ioctl(vcpufd, KVM_SET_REGS, &regs);
```

レジスタ rip は CS の先頭からのオフセットです。KVM_SET_SREGS で CS は 0x00000000 から始まるように設定したので、rip も 0 を設定しておくことで、VCPU は VM 起動後、0x00000000 の命令から実行を始めるようになります。

レジスタ rflags は CPU の状態を示すフラグです。予約ビットで 1 を書くことが決められているビットを除き、すべてのビットを 0 で初期化します。

ここまでで VM のセットアップは完了です。

1.1.8 実行

KVM_RUN リクエストを VCPU に対して発行すると VM の実行が始まります (リスト 1.10)。

リスト 1.10 01_hello/main.c

```

/* 実行 */
unsigned char is_running = 1;
while (is_running) {
    ioctl(vcpufd, KVM_RUN, NULL);

    /* 何かあるまで返ってこない */

```

KVM_RUN の ioctl は、特権命令の実行や、IO の処理などがあるまで帰ってきません。帰ってきたらリスト 1.11 のように帰ってきた理由 (exit_reason) を確認し対応する処理を行います。

リスト 1.11 01_hello/main.c

```

switch (run->exit_reason) {    /* 何かあった */
case KVM_EXIT_HLT:            /* HLT した */
    /* printf("KVM_EXIT_HLT\n"); */
    is_running = 0;
    fflush(stdout);
    break;

case KVM_EXIT_IO:             /* IO 操作 */
    if (run->io.port == 0x01
        && run->io.direction == KVM_EXIT_IO_OUT) {
        putchar(*(char *) (
            (unsigned char *)run + run->io.data_offset));
    }
}
}

```

KVM_EXIT_HLT の case は単に、標準出力をフラッシュして while ループを抜けるだけです。

KVM_EXIT_IO は、対象の IO が 0x01 であり、かつアクセスが書き込み (KVM_EXIT_IO_OUT) である場合、渡された文字を画面表示しています。これは、「シリアルポートの送信レジスタが IO アドレス空間のアドレス 1 番にある」という状態です。

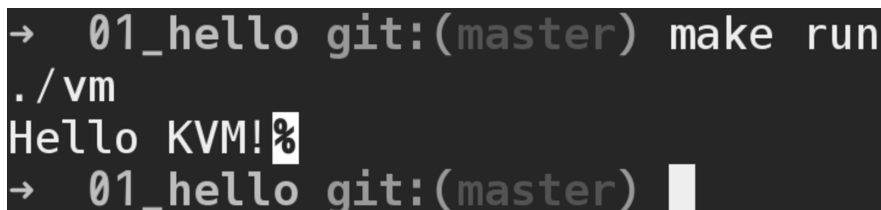
IO 処理をユーザ側で実装する際、カーネル側で動作している KVM との値の受け渡しにも VCPU をマップした領域を使います。

マップした領域の何処を使うのかを示すオフセットが run->io.data_offset で、マップした領域の先頭からのオフセットが書かれています。

今回の場合、run 変数に格納されたアドレスに run->io.data_offset を足したアドレスを参照することで、VM 上で動くプログラムが IO アドレス 1 番へ書き込んだデータを取得できます。

1.1.9 動作確認

実行すると図 1.3 のように"Hello KVM!"の文字列が表示されることを確認できます。



```
→ 01_hello git:(master) make run
./vm
Hello KVM!%
→ 01_hello git:(master)
```

図 1.3 01_hello の実行結果

"Hello KVM!"の後に '%' が付いているのは、最後に改行文字が無い印として私の使っている zsh がつけているものです。他の一般的なシェルでは"Hello KVM!"に続いてシェルのプロンプトが表示されます。

1.2 BIOS を動かす

前項で /dev/kvm へ ioctl を発行することで VM を作成し、その上で"Hello KVM!"を出力するプログラムを動作させてみました。

次は x86 PC において電源を入れてから一番最初に実行されるソフトウェアとして BIOS を動作させてみます。

1.2.1 サンプルコード

この項のサンプルコードはリポジトリ内の「02_bios」ディレクトリに格納しています。GitHub 上の URL としては以下の場所です。

- https://github.com/cupnes/bare_metal_kvm/tree/master/02_bios

また、エラー処理など削除してコードの見通しを良くしたものを「02_bios_nodebug」ディレクトリに格納しています。GitHub 上の URL としては以下です。

- https://github.com/cupnes/bare_metal_kvm/tree/master/02_bios_nodebug

前項同様、make でビルドし、make run で実行できます。

実行する際、BIOS として SeaBIOS を使うので、インストールされていない場合はインストールしておいてください。

```
$ sudo apt install seabios
```

1.2.2 この項でやること

この項では以下を行います。

- VM に最低限必要なハードウェアを設定して SeaBIOS を動かす
- SeaBIOS が動いている様子をデバッグ用のシリアルポートで確認できれば OK とする

この項のサンプルのアーキテクチャは図 1.4 の通りです。

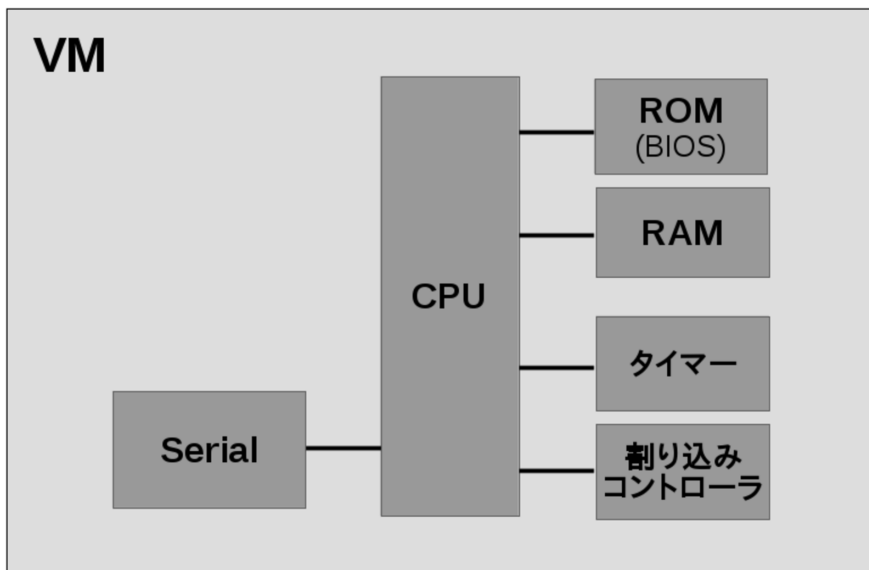


図 1.4 BIOS を動かす VM のアーキテクチャ

また、メモリマップは図 1.5 の通りです。

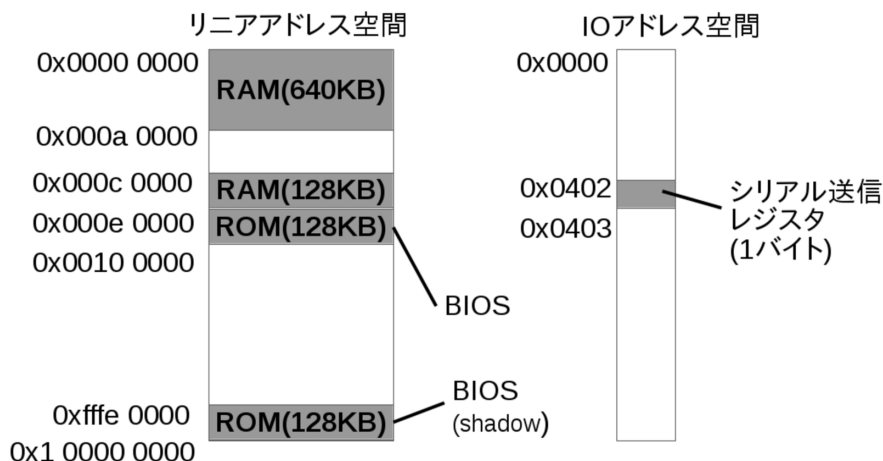


図 1.5 メモリマップ

1.2.3 SeaBIOS を動作させるために必要なこと

前項で、KVM で VM を作る際には IO 命令等に対する挙動を自分で記述する事を紹介しました。

完全に自分独自のハードウェアを VM で作ってみるのであれば、前項のように「シリアルポートは IO アドレス空間の 1 番にする」と決めて作っていくのも面白いです。

ただ、本書の場合はオープンソースの BIOS である SeaBIOS を動作させてみます。

SeaBIOS を動作させるために VM 側で用意する必要があるものは以下の 5 つです。

- 割り込みコントローラ
- タイマー
- ROM
- RAM
- デバッグ用シリアルポート

そのため、次節からこれらを VM へ追加する方法を紹介します。

1.2.4 割り込みコントローラを追加する

まず、割り込みコントローラを追加します。

KVM では CPU の他にいくつかの機能は KVM 側 (カーネル側) で持っています。

割り込みコントローラも KVM 側で持っている機能で、リスト 1.12 のように設定します。

リスト 1.12 02_bios/main.c

```
/* 割り込みコントローラ作成、VM へ設定 */
r = ioctl(vmfd, KVM_CREATE_IRQCHIP);
assert(r == 0, "KVM_CREATE_IRQCHIP");
```

KVM_CREATE_IRQCHIP という ioctl を発行するだけで、vmfd で指定した VM に割り込みコントローラ (IRQ Chip) を割り当ててくれます。

1.2.5 タイマーを追加する

タイマーも割り込みコントローラと同様です。リスト 1.13 のように ioctl を発行します。

リスト 1.13 02_bios/main.c

```
/* タイマー作成、VM へ設定 */
r = ioctl(vmfd, KVM_CREATE_PIT);
assert(r == 0, "KVM_CREATE_PIT");
```

KVM_CREATE_PIT という ioctl により、vmfd で指定した VM にタイマー (PIT) を割り当ててくれます。

1.2.6 SeaBIOS 自身が入った ROM を追加する

BIOS ROM の追加は bios_rom_install という関数に分けています。

bios_rom_install 関数は、引数 path で指定された BIOS ROM バイナリ (main.c で "/usr/share/seabios/bios.bin" (BIOS_PATH 定数) を指定) を vmfd で指定された VM へ ROM として追加します。

関数内にはいくつか処理がありますが、やりたいことは「KVM_SET_USER_MEMORY_REGION という ioctl でユーザ空間側で用意したメモリ領域を VM へ割り当てる」ということで、それ以外の処理は KVM_SET_USER_MEMORY_REGION のための準備です。

それでは、bios_rom_install 関数を先頭から見ていきます。

まず、BIOS ROM ファイルを開き、ファイルサイズを取得します (リスト 1.14)。

リスト 1.14 02_bios/bios_rom.c

```
/* BIOS のバイナリを開く */
int biosfd = open(path, O_RDONLY);
assert(biosfd != -1, "bios: open");

/* BIOS バイナリのファイルサイズ取得 */
int bios_size = lseek(biosfd, 0, SEEK_END);
assert(bios_size != -1, "bios: lseek 0 SEEK_END");
r = lseek(biosfd, 0, SEEK_SET);
assert(r != -1, "bios: lseek 0 SEEK_SET");
assert(bios_size <= BIOS_MEM_SIZE, "bios: binary size exceeds 128KB.");
```

ファイルサイズ取得処理は `lseek` で一旦ファイル末尾までシークし、その際にファイルサイズを取得して、またファイル先頭まで戻す、ということをしているだけです。

なお、この項では 128KB サイズの BIOS ROM ファイルを想定しているため、ファイルサイズが 128KB を超えていた場合は `assert` でエラー終了します。seabios パッケージで BIOS ROM ファイルがインストールされるパス `/usr/share/seabios/` には `"bios-256k.bin"` というものもありますが、128KB サイズである `"bios.bin"` を使うようにしてください。

次に、`open` した BIOS のバイナリを配置する領域を確保し、確保した領域へロードします (リスト 1.15)。

リスト 1.15 02_bios/bios_rom.c

```
/* BIOS バイナリを配置する領域を確保 */
void *bios_mem;
r = posix_memalign(&bios_mem, 4096, BIOS_MEM_SIZE);
assert(r == 0, "bios: posix_memalign");

/* BIOS バイナリを確保した領域へロード */
r = read(biosfd, bios_mem, bios_size);
assert(r != -1, "bios: read");
```

ROM にしろ後述する RAM にしろ VM へ用意するメモリ領域はユーザ空間で用意してそれを `ioctl` で VM へ割り当てる、というのが KVM で VM にメモリを割り当てる流れです。

メモリの確保には `posix_memalign` という関数を使っています。これはアラインメントされたメモリの確保を行ってくれる関数で、第 2 引数にアラインメントサイズを指定しています。

ここでは 4096 バイトアラインメントされた `BIOS_MEM_SIZE`(128KB) のメモリ領域を確保し `bios_mem` へ先頭アドレスを格納しています。これは、`KVM_SET_USER_MEMORY_REGION` の仕様で、ドキュメント^{*1}にはそのよう

^{*1} <https://github.com/torvalds/linux/blob/2861952/Documentation/virtual/kvm/api.txt#>

なことは書かれていないのですが、コードには「ページサイズ (4096 バイト) アラインメントされていないアドレスが渡された場合 EINVAL(Invalid argument) を返す」ように書かれています*2。

ここまでで、ユーザ空間側で BIOS ROM の内容が書かれたメモリ領域を用意できたので、ioctl で VM へ割り当てます。

KVM_SET_USER_MEMORY_REGION の ioctl を呼び出す処理は少し行数があるので kvm_set_user_memory_region という関数名で関数化しています (リスト 1.16)。

リスト 1.16 02_bios/util.c

```
int kvm_set_user_memory_region(
    int vmfd, unsigned long long guest_phys_addr,
    unsigned long long memory_size, unsigned long long userspace_addr)
{
    static unsigned int kvm_usmem_slot = 0;

    struct kvm_userspace_memory_region usmem;
    usmem.slot = kvm_usmem_slot++;
    usmem.guest_phys_addr = guest_phys_addr;
    usmem.memory_size = memory_size;
    usmem.userspace_addr = userspace_addr;
    usmem.flags = 0;
    return ioctl(vmfd, KVM_SET_USER_MEMORY_REGION, &usmem);
}
```

KVM_SET_USER_MEMORY_REGION の ioctl は、「VM へ割り当てたいユーザ空間側で用意した領域の先頭アドレス」や「VM 上のどこのアドレスへ割り当てるか」といった情報を「struct kvm_userspace_memory_region」という構造体のポインタで渡します。そのため、構造体変数を作って、構造体のメンバへ値を設定し、ioctl 時に引数に渡す、ということを行っています。

kvm_set_user_memory_region 関数を使って、ユーザ空間で用意した BIOS ROM のメモリ空間を VM へ割り当てます (リスト 1.17)。

リスト 1.17 02_bios/bios_rom.c

```
/* BIOS をロードした領域を VM へマップ (legacy) */
r = kvm_set_user_memory_region(vmfd, BIOS_LEGACY_ADDR, BIOS_MEM_SIZE,
                                (unsigned long long)bios_mem);
assert(r != -1, "bios: KVM_SET_USER_MEMORY_REGION(legacy)");

/* BIOS をロードした領域を VM へマップ (shadow) */
r = kvm_set_user_memory_region(vmfd, BIOS_SHADOW_ADDR, BIOS_MEM_SIZE,
```

L1005

*2 https://github.com/torvalds/linux/blob/2861952/virt/kvm/kvm_main.c#L940

```
(unsigned long long)bios_mem);
assert(r != -1, "bios: KVM_SET_USER_MEMORY_REGION(shadow)");
```

同じ BIOS ROM 領域を VM 上の BIOS_LEGACY_ADDR(0x000e0000) と BIOS_SHADOW_ADDR(0xfffe0000) という 2 つの領域へ割り当てています。

これは、電源投入直後の CPU の動作と BIOS が慣例的に割り当てられるアドレス領域によるものです。電源投入直後、CPU は 0x...ffff0 という最後の 4 ビットを 0 にしたアドレスから命令を読み込んで実行します。しかし、Intel の CPU は互換性のために電源投入直後は古い CPU の設定で立ち上がるため、起動直後はアドレスの 20 ビット目以上が全て無効 (0) になります。そのため、実際には 0xffff0 から命令を読み込んで実行することになります。このアドレスは古くから BIOS が配置されているアドレス (0xe0000 - 0xfffff) であり、問題なく BIOS が実行されます。

ただし、0xffff0 から実行を開始する場合、アドレスの末尾である 0xffff までには 16 バイトしかありません。そのため、0xffff0 から 16 バイトの領域には BIOS の空間の先頭へジャンプする命令が書かれています。

ここまでが、実機で BIOS が実行開始されるまでの一般的な流れです。KVM の VCPU の場合、実はこのような流れにはなりません。

KVM の VCPU の場合に (少なくとも特に設定をしないで使う場合は、)「アドレスの 20 ビット目以上を全て無効にする」というようなことはせず、起動直後は 0xfffffff0 から実行します。そのため、0xfffe0000 - 0xfffffff0 にも BIOS を配置し、0xfffffff0 から実行を開始された場合も 0xe0000 からの BIOS の空間へジャンプできるようにしています^{*3}。

1.2.7 RAM を追加する

SeaBIOS が動作する上で必要な RAM を用意します (リスト 1.18)。

リスト 1.18 02_bios/main.c

```
#define RAM1_BASE      0x00000000
#define RAM1_SIZE      0x000A0000 /* 640KB */
#define RAM2_BASE      0x000C0000 /* VGA BIOS Base Address */
#define RAM2_SIZE      0x00020000 /* 128KB */

...

/* RAM 作成、VM へ設定 */
ram_install(vmfd, RAM1_BASE, RAM1_SIZE);
ram_install(vmfd, RAM2_BASE, RAM2_SIZE);
```

^{*3} もちろん、前項のように CPU の実行開始アドレスを明示的に指定すればこのような面倒なことをしなくても良いです。

VM 上に用意する RAM の領域は 2 つで、VM 上のベースアドレスとサイズは定数の通りです。

VM への RAM 割り当ては `ram_install` という名前で関数化しています (リスト 1.19)。

リスト 1.19 02_bios/ram.c

```
void ram_install(int vmfd, unsigned long long base, size_t size)
{
    int r;
    void *addr;

    r = posix_memalign(&addr, 4096, size);
    assert(r == 0, "ram: posix_memalign");

    r = kvm_set_user_memory_region(vmfd, base, size,
                                   (unsigned long long)addr);
    assert(r != -1, "ram: KVM_SET_USER_MEMORY_REGION");
}
```

指定されたサイズのメモリを確保して `kvm_set_user_memory_region` 関数で VM へ割り当てています。メモリ確保や VM 割り当ての方法は ROM の時と同じです。

1.2.8 デバッグ用シリアルポートを用意する

SeaBIOS は自身の動作ログを 0x0402 という IO アドレスへ 1 文字 (1 バイト) ずつ出力します。そこで、前項で IO アドレス 1 番としていたものを 0x0402 番へ変更します。

KVM_RUN を IO 要因で Exit した場合に IO をハンドリングする処理は `io_handle` という関数へ分けてみました (リスト 1.20、リスト 1.21)。

リスト 1.20 02_bios/main.c

```
/* VM 実行 */
while (1) {
    DEBUG_PRINT("Enter: KVM_RUN\n\n");
    r = ioctl(vcpufd, KVM_RUN, 0);
    assert(r != -1, "KVM_RUN");
    DEBUG_PRINT("Exit: KVM_RUN(0x%08x)\n", run->exit_reason);

    dump_regs(vcpufd);

    switch (run->exit_reason) {
    case KVM_EXIT_IO:
        io_handle(run);
        break;

    default:
```

```
        fflush(stdout);
        assert(0, "undefined exit_reason\n");
    }
}
```

リスト 1.21 02_bios/io.c

```
void io_handle(struct kvm_run *run)
{
    DEBUG_PRINT("io: KVM_EXIT_IO\n");
    DEBUG_PRINT("io: direction=%d, size=%d, port=0x%04x,",
                run->io.direction, run->io.size, run->io.port);
    DEBUG_PRINT(" count=0x%08x, data_offset=0x%016llx\n",
                run->io.count, run->io.data_offset);

    if (run->io.port == SERIAL_IO_TX)
        serial_handle_io(run);
}
```

1.2.9 動作確認

実行すると図 1.6 のように SeaBIOS の動作ログが表示され、SeaBIOS が実行されている様子を確認できます。


```

+ 02_bios git:(master) make run
./vm
SeaBIOS (version 1.10.2-1)
BUILD: gcc: (Debian 6.3.0-11) 6.3.0 20170321 binutils: (GNU Binutils for Debian) 2.28
Unable to unlock ram - bridge not found
RamSize: 0x00100000 [cmos]
Relocating init from 0x000e3160 to 0x00081d80 (size 57920)
=== PCI bus & bridge init ===
Detected non-PCI system
No apic - only the main cpu is present.
Copying PIR from 0x0008fd00 to 0x000f1320
Copying MPTABLE from 0x00006e90/79cc0 to 0x000f1250
WARNING - Unable to allocate resource at smbios_legacy_setup:516!
Scan for VGA option rom
WARNING - Timeout at i8042_wait_read:38!
ATA controller 1 at 1f0/3f4/0 (irq 14 dev ffffffff)
ATA controller 2 at 170/374/0 (irq 15 dev ffffffff)
Found 0 lpt ports
Found 0 serial ports
Scan for option roms

Press ESC for boot menu.

Searching bootorder for: HALT
Space available for UMB: c0000-ef000, f0000-f1210
e820 map has 4 items:
  0: 0000000000000000 - 0000000000009fc00 = 1 RAM
  1: 0000000000009fc00 - 00000000000a0000 = 2 RESERVED
  2: 00000000000f0000 - 0000000000100000 = 2 RESERVED
  3: 00000000fffc0000 - 0000000100000000 = 2 RESERVED
Unable to lock ram - bridge not found
enter handle_19:
  NULL
Booting from Floppy...
Boot failed: could not read the boot disk

```

図 1.6 02_bios の実行結果

SeaBIOS 自体を動かすことはできましたが、動いた結果としてはフロッピーディスクやハードディスク等からのロードに失敗、という状態です。

それではディスクを扱えるように、まずはハードディスクに比べて簡単なフロッピーディスクから VM 上に実装していきたい所です。そのためにはまず PCI を扱えるようにする必要があり、PCI を BIOS で認識するためには PCI コンフィグレーション空間という PCI の設定値が格納された領域を VM 側に用意し、BIOS からはそれを IO 命令で読み出せるようにする必要があり、となります。

それらをコツコツと作っていても良いのですが、本書の場合、自作 OS 自動テストの第一歩である「MBR のテスト」を実現することが目的で、スクラッチで 1 から作っていくことを目的とはしていません。KVM を扱う既存のコードを改造するための知識としてはひとまずこちら辺で十分^{*4}なので、次の章からは早速、既存の比較的手軽なコードを改

^{*4} PCI コンフィグレーション空間もそうですが、引き続き作っていく場合、今後は BIOS が求める IO を KVM_RUN から戻ってきた後の IO ハンドラに実際の PC と同様のものを一つ一つ実装していく作業です。デバイス毎の違いはあれど、VM がその上で動くプログラムへ IO の結果を返したり、逆に IO の

造して「MBR 自動テスト」を実現してみます。

出力を受け取ったりする方法はシリアルポートで説明した通りなので、同じような作業を繰り返すことになります。

第 2 章

既存のコードを改造して MBR テスターを作る

前章までで KVM を使う VM はどのように作られているかを実際に簡単な VM を作ることで確認しました。

この章では、既存の扱いやすいコードを元に MBR 自動テストを実現します。

なお、この章から MBR をテストするプログラムを「MBR テスター」と呼ぶことにします。

2.1 実現すること

この章で作る MBR テスターとしては、最低限以下が実現できていれば良いこととします。

- MBR が特定の文字列を出力するか否かをテストする
- テストを pass すると、MBR テスターは終了ステータス 0 を返す。テストを fail すると、終了ステータス 1 を返す

2.2 手軽そうなサンプルを探す

それでは、前述した内容を実現していきます。

前章の最後に説明した通り、全て自作するのではなく、本書では手軽に使いそうなサンプルを探してそれを改造することで目的を実現してみます。

`/dev/kvm` を直接叩くようなプログラムとしては QEMU が一番有名ですが、QEMU は全コード量として 160 万行ほどあり (2018 年 9 月現在)、コンパイルするだけでも大変

です。

そこで、「こういうことをやっているコード量が少なくてコンパイル&実行可能なサンプル無いかな」と探するとき、GitHub のコード検索はとても便利です。

<https://github.com/search> にアクセスして、検索窓に含んでほしいコード片を書いて検索すると、GitHub 内の全リポジトリから該当するソースコードを一覧表示してくれます。

「advanced search」を使うと「Advanced options」の「Written in this language」から言語の指定や、「Code options」の「Of this file size」からファイルサイズ(おそらくコード行数?)の指定もできます。

2.3 kvmulate について

「/dev/kvm を直接使っていて」、「seabios のバイナリも直接使っていて」、「MBR をロードして実行する」様なサンプルとしては、「kvmulate」というリポジトリのコードが良さそうです。

- <https://github.com/slowcoder/kvmulate>

全コード量も 5071 行と QEMU に比べ圧倒的に少なく、main 関数から処理を追っていった全体を把握するのもそんなに大変ではありません。

これから kvmulate を改造していく上で、この項では kvmulate のコンパイル&動作確認と、kvmulate のソースコードを簡単に概観してみます。

2.3.1 コンパイル

まず、kvmulate のソースコードを git clone、あるいは GitHub のページから zip でダウンロードするなどしてローカルへ配置してください。

kvmulate は、Simple DirectMedia Layer(SDL) という GUI 表示するために使っているライブラリを除いて、標準的な C のライブラリと make しか使わないです。そのため、本書で想定している apt が使える Linux 環境であれば、以下の 2 つのパッケージがインストールされていれば問題なくコンパイルできると思います。

- build-essential
- libsdl1.2-dev

Makefile が用意されているので、kvmulate のディレクトリ内で"make"を実行すればコンパイルできます。

2.3.2 MBR を作成

テスト対象である MBR を用意します。

kvmulate は、同じディレクトリ内に"floppy.img"や"hdd.img"というファイル名でイメージファイルを置いておけば、それらをフロッピーディスクやハードディスクの生のイメージとして使います。

そこで、512 バイトのバイナリイメージを MBR として作成し、それらを"floppy.img"や"hdd.img"というファイル名で置いておくことにします。

MBR としては、自身で BIOS の機能呼び出して画面に文字を出力するようなプログラムを作成します。

サンプルは 1 章で紹介したリポジトリの「mbr_sample」ディレクトリに格納していません。URL は以下の通りです。

- https://github.com/cupnes/bare_metal_kvm/tree/master/mbr_sample

プログラムは短めのアセンブラです (リスト 2.1)。

リスト 2.1 mbr_sample/output_A.s

```
.code16

movb    $0x41, %al
movb    $0x0e, %ah
int     $0x10

jmp     .
```

".code16"はアセンブラへ 16 ビット向けのソースコードであることを示す指定です。BIOS は CPU が 16 ビットのモード (リアルモード) で動作するため、アセンブラも 16 ビット向けで記述します。

次の 3 行からなるコードブロックで、BIOS の機能呼び出して 'A' という文字を画面へ描画しています。BIOS は「Basic Input/Output System」という名前の通り、基本的な I/O のシステムを持っています。

BIOS の機能呼び出す方法はソフトウェア割り込みで、機能ごとに割り込み番号が決められています。そしてパラメータの受け渡しには汎用レジスタを用い、機能ごとにどの汎用レジスタで何の受け渡しを行うかが決められています。

今回の場合、割り込み番号 0x10 が画面制御機能群で、AH レジスタにその中でどんな機能呼び出すかを指定します。0x41 が ASCII で文字出力を行う機能です。AL レジスタへは出力したい文字コード (今回の場合 'A'(0x41)) を ASCII で指定します。

最後の"jmp ."は無限ループで、このプログラムの戻り先は無いのでここで止めています。"."は「今この場所」を示すもので、jmp の先として指定すると「今この場所へジャンプする」事を繰り返すため無限ループになります。

他の文字を出力させたり、BIOS の別の機能を呼び出したい場合は、リスト 2.1 を書きかえてみてください。

Makefile とリンカスクリプト (mbr.ld) も用意しているので、mbr_sample ディレクトリで"make"を実行するだけで MBR のバイナリ (output_A.mbr) を生成できます。

2.3.3 実行するには

kvmulate は、実行する際、以下の 5 つが同じディレクトリに配置されている必要があります。

- kvmulate
- seabios.rom
- vgabios.rom
- floppy.img
- hdd.img

「seabios.rom」と「vgabios.rom」は、kvmulate のリポジトリに含まれています^{*1}。

また、「floppy.img」と「hdd.img」は共に同じもので構いません。本書では、MBR のバイナリ (例えば、前節で作った output_A.mbr) を「floppy.img」と「hdd.img」という 2 つのファイルへコピーして配置しています^{*2}。

2.3.4 kvmulate のコード構成

kvmulate は図 2.1 の構成になっています^{*3}。

^{*1} もちろん、APT でインストールした SeaBIOS に含まれるバイナリを使っても構いません。その際は、たいてい/usr/share/seabios/にインストールされていますので、この中にある「bios.bin」を「seabios.rom」へ、「vgabios-stdvga.bin」を「vgabios.rom」へリネームして使ってください。

^{*2} kvmulate 内の処理として、floppy.img を使用する場合でも hdd.img のファイルチェックをするコードパスに入るためです。hdd.img を都度用意するのが面倒であれば、「hdd.img」を open しているところ辺りの処理を削除すれば良いです。

^{*3} 本書の改造の範囲外 (DMA など) は省略しています。

kvmulate

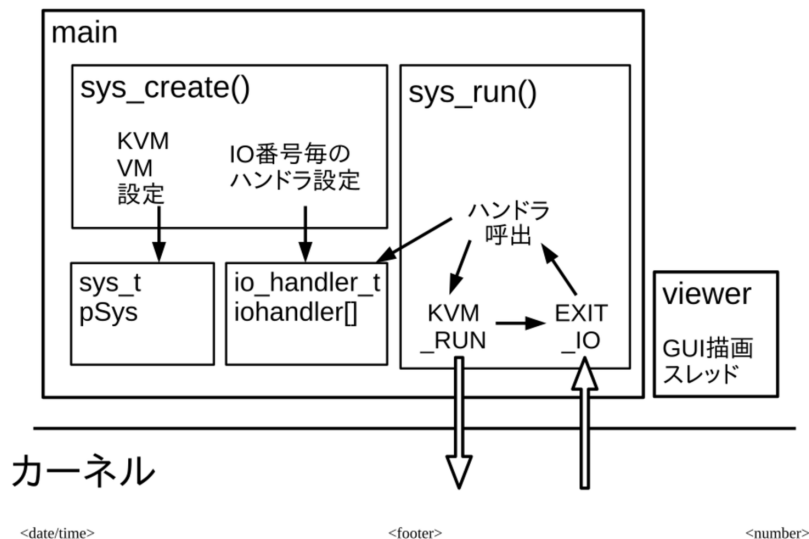


図 2.1 kvmulate の構成

`system.c` に定義されている `sys_create` 関数 (とそこから呼び出される関数) で `KVM_RUN` 実行前の準備 (VM 作成) を行い、同じく `system.c` に定義されている `sys_run` 関数 (とそこから呼び出される関数) で `KVM_RUN` と `KVM_RUN` から戻ってきた時のハンドリングを行います。

`sys_create` 関数からは、`sys_t` という構造体の型の `pSys` へ VCPU と VM の設定を行い、`io_handler_t` という同じく構造体型の `iohandler` 配列へ IO ポート毎のハンドラ設定を行います。

また、`sys_run` 関数からは `KVM_RUN` を行い、IO で `EXIT` してきた際には `iohandler` 配列の該当する IO ポートの要素を参照して登録されているハンドラを呼び出します。

2.4 改造する

それでは、`kvmulate` を改造して MBR テスターを実現してみます。

以降では変更箇所を説明しますが、`patch` ファイル (diff) がほしい場合は以下の筆者のページに Gist へのリンクを掲載しますので、参照してみてください。

- <http://yuma.ohgami.jp>

主に変更するファイルは Makefile(リスト 2.2) と GUI 表示を行う viewer.c(リスト 2.3) です。あと、kvmulate の標準出力へのログ出力等を抑制します。

リスト 2.2 kvmulate/Makefile

```
DEPS := $(subst .o,.d,$(OBJS))
SRCS := $(subst .o,.c,$(OBJS))

all: mbr_tester # 変更

mbr_tester: $(OBJS) # 変更
    $(CC) $(CFLAGS) -pthread -o $@ $^ # -lSDLを外す

%.o: %.c
    @echo "CC      $<"

clean:
    @echo "CLEAN"
    @$ (RM) -r $(OBJS) $(DEPS) mbr_tester boot.bin # 変更
```

SDL を外す他に、生成する実行ファイル名も "mbr_tester" へ変更しました。

viewer.c については SDL の処理をがっつりと消してしまい、リスト 2.3 のようにしました。

リスト 2.3 kvmulate/devices/x86/vga/viewer.c

```
#include <unistd.h>
#include <pthread.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "devices/x86/vga/vga.h"
#include "devices/x86/vga/font.h"
#include "log.h"

static uint8 test_ch(uint8 ch)
{
    return ch == 'A';
}

#define TEST_FAILURE_TH 200
static void updateTextmode(vgactx_t *pVGA) {
    int x,y;
    uint8 ch,attr;
    uint32 vram_off;

    vram_off = (pVGA->crtc_reg[0xC] << 8) | pVGA->crtc_reg[0xD];
    vram_off *= 2;
```



```

static uint32 test_counter = 1;
for(y=0;y<25;y++) {
    for(x=0;x<80;x++) {
        ch = pVGA->pVRAM[0][0x18000 + vram_off + (y*80+x)*2+0];
        attr = pVGA->pVRAM[0][0x18000 + vram_off + (y*80+x)*2+1];

        if (y > 4) {
            if (test_ch(ch)) {
                printf("pass(test_counter=%d)\n", test_counter);
                exit(EXIT_SUCCESS);
            } else if (test_counter > TEST_FAILURE_TH) {
                printf("fail(test_counter=%d)\n", test_counter);
                exit(EXIT_FAILURE);
            }
        }
    }
}
test_counter++;
}

static void *viewer_thread(void *pArg) {
    vgactx_t *pVGA = (vgactx_t*)pArg;
    int bDone = 0;

    LOG("pVRAM[0] = %p", pVGA->pVRAM[0]);

    while(!bDone) {
        updateTextmode(pVGA);

        usleep(1000000LL / 60LL);
    }

    return NULL;
}

int x86_vga_viewer_init(vgactx_t *pVGA) {
    pthread_t thid;

    LOG("pVRAM[0] = %p", pVGA->pVRAM[0]);
    pthread_create(&thid, NULL, viewer_thread, pVGA);

    return 0;
}

```

viewer.c の処理が呼び出される流れは以下の通りです。

1. sys_create 関数が、IO ハンドラの初期化などのために devices_register_all 関数 (devices/devices.c) を呼び出す
2. devices_register_all 関数から viewer.c の x86_vga_viewer_init 関数が呼ばれる
3. x86_vga_viewer_init 関数はスレッドを生成し、viewer_thread 関数を実行させる
4. viewer_thread 関数は 60fps で画面更新 (updateTextmode 関数) を行う
5. updateTextmode 関数では、文字出力等により VRAM の領域 (pVGA->pVRAM)

に格納された文字を描画する(ただし、描画処理は丸ごと削っている)

そのため、updateTextmode 関数で、描画しようとしている文字を拾えば、出力される文字のテストができます。

ここでは、test_ch という関数を追加して、MBR が"A"という文字を出力することをテストしています。5 行目以降 ($y > 4$) の文字をテスト対象にしているのは、4 行目までは BIOS の出力があるからです。

また、変数 test_counter は、updateTextmode 関数呼び出しの回数、すなわち 60fps での画面描画回数です。定数 TEST_FAILURE_TH(200) の回数を超えても tech_ch 関数を pass しないようであれば、テストは fail と判定します。

最後に、標準出力へのログ出力等を抑制します(リスト 2.4、リスト 2.5)。

リスト 2.4 kvmulate/log.c

```
#include <stdio.h>
#include <string.h>
#include <stdarg.h>

#include "log.h"

//int LogLevel = LOGLEVEL_INFO;
//int LogLevel = LOGLEVEL_DEBUG;      /* コメントアウト */
int LogLevel = -1;    /* 追加 */
```

リスト 2.5 kvmulate/devices/x86/bios_debug.c

```
static void biosdebug_outb(struct io_handler *hdl, uint16 port, uint8 val) {
    /* putchar(val); */    /* コメントアウト */
}
```

リスト 2.4 は kvmulate のログ出力で、LogLevel が-1 だと何も出力されなくなります。

リスト 2.5 について、biosdebug_outb 関数は、SeaBIOS がデバッグ情報を出力する際の IO 命令 (IO アドレス 0x0402) のハンドラです。これも関数内の putchar 関数をコメントアウトすることで抑制しています。

なお、MBR から "int 0x10" で BIOS の機能呼び出して出力する文字については BIOS のデバッグ出力では出力されません。そのため、先述した「5 行目以降に MBR が出力させる文字が表示される」という確認は、viewer.c の updateTextmode 関数で標準出力に行番号付きで文字を出力させると分かりやすいです。本章最後の補足にまとめているので興味があればご覧ください。

2.5 動作確認

それでは、動作させてみます。コンパイルし、生成された"mbr_tester"バイナリを実行すると、以下のようにテストが通り、終了ステータス 0 で終了できていることを確認できます。

```
$ ./mbr_tester && echo 'kvmulate is succeeded.' || echo 'kvmulate is failed.'
pass(test_counter=141)
kvmulate is succeeded.
```

また、試しに sample_A の出力文字、あるいは mbr_tester 側のテスト内容のいずれかのみを変更すると、以下のようにテストが失敗し、終了ステータスが 0 以外で終了できていることを確認できます。

```
$ ./mbr_tester && echo 'kvmulate is succeeded.' || echo 'kvmulate is failed.'
fail(test_counter=201)
kvmulate is failed.
```

2.6 補足: GUI 上の内容を標準出力へ出す

デバッグ用途も兼ねて、GUI 画面上に表示されていたメッセージ等を行番号付きで標準出力へ出力させるには、viewer.c の updateTextmode 関数をリスト 2.6 のように改造すれば良いです。

リスト 2.6 devices/x86/vga/viewer.c

```
static void updateTextmode(vgactx_t *pVGA) {
    int x,y;
    uint8 ch,attr;
    uint32 vram_off;

    vram_off = (pVGA->crtc_reg[0xC] << 8) | pVGA->crtc_reg[0xD];
    vram_off *= 2;

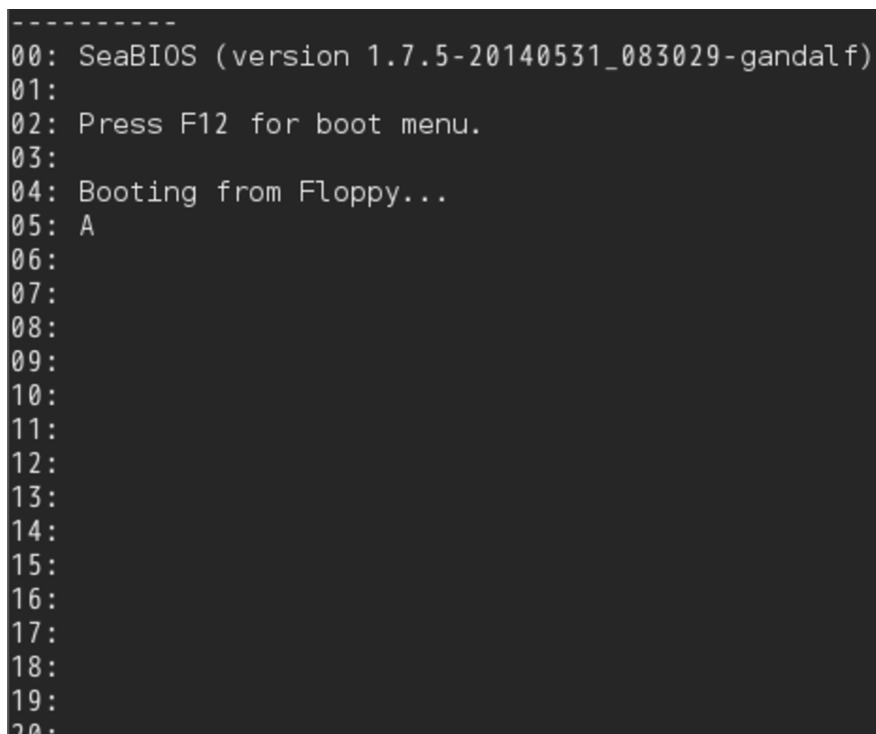
    static uint32 test_counter = 1;
    printf("\x1b[2J"); /* 追加 */
    printf("-----\n"); /* 追加 */
    for(y=0;y<25;y++) {
        printf("%02d: ", y); /* 追加 */
        for(x=0;x<80;x++) {
            ch = pVGA->pVRAM[0][0x18000 + vram_off + (y*80+x)*2+0];
```

```
attr = pVGA->pVRAM[0][0x18000 + vram_off + (y*80+x)*2+1];

/* test_ch 関数呼び出し周りの処理は削除 */

    putchar(ch);      /* 追加 */
}
    putchar('\n');    /* 追加 */
}
    printf("-----\n"); /* 追加 */
}
```

これで「'A' だけを出力する」MBR を実行すると図 2.2 のようになります。



```
-----
00: SeaBIOS (version 1.7.5-20140531_083029-gandalf)
01:
02: Press F12 for boot menu.
03:
04: Booting from Floppy...
05: A
06:
07:
08:
09:
10:
11:
12:
13:
14:
15:
16:
17:
18:
19:
20:
```

図 2.2 GUI の内容をシェル上へ出力するよう改造

このように、 $y=5$ 行目以降に MBR で出力させる文字が表示されるため、前述の `test_ch` 関数は $y>4$ の条件で呼び出していました。

第3章

遺伝的 MBR を実現する

3.1 遺伝的アルゴリズムについて

本書で扱う遺伝的アルゴリズムの流れを説明します。一般的な定義と大きく違うところは無いと思いますが、筆者は遺伝的アルゴリズムに詳しいわけでは無いので、中には本書独自のところもあるかと思います。

本書では MBR を遺伝的アルゴリズムにおける一つ一つの個体として扱い、遺伝的アルゴリズムを適用してみます。この場合、MBR を構成するバイナリ列が「遺伝子^{*1}」となります。

遺伝的アルゴリズムは大きく以下の流れで行います。

1. 初期個体群生成
2. 評価
3. フィードバック
4. 以降、2 と 3 を繰り返す

それぞれについて簡単に説明します。(詳細は後述します。)

「1. 初期個体群生成」は、単に MBR のバイナリ列を個体数分用意するだけです。

「2. 評価」で前章で作った MBR テスターを使います。次のステップのために各個体に対する評価結果を保存しておきます。

「3. フィードバック」では評価結果を個体群へフィードバックします。まず評価結果の悪かった個体を淘汰し、優秀な個体はそのままコピーして増やします。そして、上位何割かで遺伝子を交叉させて子を作り、また何割かで遺伝子の突然変異を起こします。

^{*1} 「遺伝子」という言葉は、本来、DNA の中でタンパク質の作り方を記録している部分を指しますが、本書では(というより遺伝的アルゴリズムの世界では?)「遺伝に関する情報」や「各個体を形作るデータ列」といった意味で「遺伝子」という言葉を使っています。

以降、2 と 3 を繰り返すことで目的の MBR へ近づけていきます。なお、2 と 3 を行う 1 サイクルを「世代」と呼ぶことにします。「評価とフィードバックが終わると現在の世代が終わり、次のループから次の世代が始まる」というイメージです。

3.2 実装について

本書では、完全に筆者の趣味で、シェルスクリプトで実装しています。以下のリポジトリで公開しています。

- https://github.com/cupnes/ge_mbr

ただ、シェルスクリプトは一般的にあまり読みやすいものではないです。そこで、ここでは先述した大きく 3 つの処理それぞれについて、どのように実装したのかを説明していくことにします。

そんなに大したものではないので、もし興味があれば自身のお好きな言語で実装してみることをおすすめします。

3.2.1 スクリプト全体の処理の流れ

リポジトリ内の "ge_mbr.sh" に全て書かれています。

先述した 3 つの処理は関数化していますので、それらの説明は後ですとして、ここでは全体の処理をざっと説明します。「実装の詳細は置いておいて、ひとまず遊んでみたい」という場合もこの節さえ読んでもらえれば大丈夫です。

まず、グローバル変数を定義します (リスト 3.1)。

リスト 3.1 グローバル変数定義

```
#!/bin/bash

# set -eux

# POPULATION_SIZE は、
# - 2 で 2 回割っても余りが出ない
# - 5 で 1 回割っても余りが出ない
# を共に満たす値であること
# すなわち、20 の倍数
POPULATION_SIZE=20
GENE_LEN=446
MBR_TESTER=./mbr_tester
WORK_DIR=$(date '+%Y%m%d%H%M%S')

# ...以降、関数定義が続く...
```

"POPULATION_SIZE" は個体数です。後の処理の都合上、コメントの通り 20 の倍数

である必要があります。

GENE_LEN は遺伝子の長さ (バイト数) です。MBR512 バイトは、「イニシャルプログラムローダ (IPL)」(446 バイト) と「パーティションテーブル」(64 バイト) と「ブートシグネチャ」(2 バイト) で構成されています。遺伝的アルゴリズムで求めたいのは BIOS によってロードされる実行バイナリ部分である IPL なので、446 を遺伝子の長さとしています。とはいえ、MBR テスターで実行する際は 512 バイト揃っていないといけないため、パーティションテーブルとブートシグネチャのバイナリは "mbr_partition_tbl_boot_sig.dat" というバイナリでリポジトリ内に予め用意しており、IPL のバイナリに結合して 512 バイトの MBR を作ります。

WORK_DIR は作業ディレクトリです。この中に全固体の全世代のデータが保存されていきます。

関数定義部分は飛ばして、次はコマンドライン引数チェックです (リスト 3.2)。

リスト 3.2 コマンドライン引数チェック

```
if [ $# -ge 1 ]; then
    WORK_DIR=$1
    echo "Use WORK_DIR=${WORK_DIR}"
else
    echo "Initialization"
    initialization
fi

if [ $# -ge 2 ]; then
    age=$2
    echo "It will start in the generation $age."
else
    age=0
fi
```

ge_mbr.sh は途中再開のための引数を最大 2 つとります。

```
$ ./ge_mbr.sh [作業ディレクトリ] [世代番号]
```

それぞれの引数が指定されれば、指定されたディレクトリで指定された世代番号から再開します。

指定されなかった場合は、作業ディレクトリは先述したディレクトリで、世代番号 0 から始めます。

次に、メイン処理が続きます (リスト 3.3)。

リスト 3.3 メイン処理

```
touch enable_ge
while [ -f enable_ge ]; do
    echo "GE: age=$age"
    echo '-----',

    evaluation
    feedback
    mv ${WORK_DIR}/{now,$age}
    mv ${WORK_DIR}/{next,now}
    mkdir ${WORK_DIR}/next
    age=$((age + 1))

    echo
    echo
done
```

「評価 (evaluation 関数)」と「フィードバック (feedback 関数)」を while で繰り返しています。

while の直前で touch により空ファイルを生成し、while のループ条件にもなっている "enable_ge" ファイルは、ge_mbr.sh スクリプトを中断させるためのファイルです。evaluation 関数や feedback 関数では作業ディレクトリ内で適宜ファイルを置いたり移動させたりしながら処理を行います。これらの作業中に Ctrl+C 等で終了させてしまうと作業ディレクトリ内が中途半端な状態になり、再開させることが難しくなります。そのため、enable_ge ファイルを用意し、このファイルを消すことで現世代の処理を終えたら while ループを抜けることができるようにしています。

評価とフィードバックが終わったところで行っているのは作業ディレクトリ内のディレクトリ名の更新です。

ここで、作業ディレクトリ内の構成について説明します。ある時点の作業ディレクトリ内はリスト 3.4 のようになっています。

リスト 3.4 ある時点の作業ディレクトリの状態

```
20180912144115/
0/
  ch_0.dat
  ch_1.dat
  ch_2.dat
  ...
  evaluation_value_list.csv
1/
  ch_0.dat
  ch_1.dat
  ch_2.dat
  ...
  evaluation_value_list.csv
now/
  ch_0.dat
```



```

ch_1.dat
ch_2.dat
...
evaluation_value_list.csv
next/

```

"ch_0.dat"や"ch_1.dat"というファイルそれぞれが 512 バイトの MBR です。"now"ディレクトリは現在の世代です。evaluation 関数での評価は"now"ディレクトリに対して行います。また、"next"ディレクトリは次の世代を示すディレクトリで、feedback 関数で次の世代の個体群を作るために使います。"0"や"1"等の数字のディレクトリは過去の世代のディレクトリです。

なお、各ディレクトリに存在する"evaluation_value_list.csv"はその世代の全固体のファイル名と評価によって得られた評価値の CSV です。evaluation 関数内で作成します。

リスト 3.3 のディレクトリ名更新処理に説明を戻すと、フィードバックまでを終えると next ディレクトリには次の世代の個体群が揃った状態になっているので、現世代の"now"ディレクトリを現在の世代番号 (\$age) ヘリネームし、次世代の"next"ディレクトリを現世代を示す"now"ディレクトリヘリネームしています。すると"next"ディレクトリが無くなるので mkdir で作成し、世代番号をインクリメントします。

最後に再開させる際のコマンドを出力します (リスト 3.5)。

リスト 3.5 スクリプト終了処理

```

echo '-----'
echo "Stopped at $(date +%Y-%m-%d %H:%M:%S)"
echo 'Continue Command'
echo "$0 ${WORK_DIR} $age"
echo '-----'

```

出力例はリスト 3.6 の通りです。

リスト 3.6 終了処理の出力例

```

-----
Stopped at 2018-09-14 15:17:15
Continue Command
./ge_mbr.sh 20180912144115 821
-----

```

この場合、再開させる際は"./ge_mbr.sh 20180912144115 821"というコマンドで再開できます。

ここまでで、スクリプト全体を俯瞰した解説は終わりです。このスクリプトを単に道具として使ってみたい場合はここらへんまでで、遊んでみてください。

先述した「初期個体生成」「評価」「フィードバック」の実装に興味がある場合は、次節からの説明をご覧ください。

3.2.2 初期個体群生成

初期個体群生成を行う `initialization` 関数はリスト 3.7 の通りです。

リスト 3.7 初期個体群生成

```
initialization() {
    local i

    echo 'initialization'
    echo '-----'

    echo "WORK_DIR=${WORK_DIR}"
    mkdir -p ${WORK_DIR}/{now,next}
    for i in $(seq 0 $((POPULATION_SIZE - 1))); do
        echo "${WORK_DIR}/now/ch_${i}.dat"
        generate "${WORK_DIR}/now/ch_${i}.dat"
    done

    echo
    echo
}
```

作業ディレクトリ内に `now` と `next` のディレクトリを作成し、`generate` 関数を呼び出して個体数分の `"ch_番号.dat"` ファイルを `now` ディレクトリに生成します。

`generate` 関数の実装はリスト 3.8 の通りで、引数で受け取ったファイル名の MBR の IPL 部分をランダムで生成するだけです。

リスト 3.8 各個体の生成

```
generate() {
    local file

    file=$1
    dd if=/dev/urandom of=$file bs=1 count=$GENE_LEN
    cat mbr_partition_tbl_boot_sig.dat >> $file
}
```

3.2.3 評価

次に、現世代の評価を行う `evaluation` 関数の実装はリスト 3.9 の通りです。

リスト 3.9 評価

```

evaluation() {
    local i
    local ch

    echo 'evaluation'
    echo '-----'

    rm -f tmp.csv

    for i in $(seq 0 $((POPULATION_SIZE - 1))); do
        echo "i=$i"
        ch=${WORK_DIR}/now/ch_$i.dat
        echo ">>>> $ch"

        cp $ch floppy.img
        cp $ch hdd.img
        $MBR_TESTER
        exit_code=$?
        case $exit_code in
            0)
                evaluation_value=100
                echo "$ch has reached the answer!"
                rm -f enable_ge
                ;;
            1)
                evaluation_value=1
                echo "$ch is an ordinary child."
                ;;
            *)
                evaluation_value=0
                echo "$ch is dead."
                ;;
        esac

        echo "ch_$i.dat,${evaluation_value}" >> tmp.csv
        echo "evaluation_value=${evaluation_value}"

        echo
        echo
    done

    # 第2列で降順にソート
    sort -n -r -k 2 -t ',' tmp.csv > ${WORK_DIR}/now/evaluation_value_list.csv

    rm -f floppy.img hdd.img tmp.csv

    echo
    echo
}

```

少し行数が多いですが、やっていることは大したことではありません。

個体数の分だけ for ループを回し、各個体の MBR ファイルで MBR テスターを実行しています。

実行後、case で終了ステータスで条件分岐しています。

0 はテストを pass し、EXIT_SUCCESS で MBR テスターが終了した場合なので、評

価値 (evaluation_value 変数) に最高の 100 を設定し、答えに到達した旨を echo しています。その後、enable_ge を削除し現世代でスクリプトを終了させるようにしています。

1 はテストが fail し、EXIT_FAILURE で MBR テスターが終了した場合なので、評価値には 1 を設定し、この個体は平凡であった旨を出力します。

それ以外は、この個体の MBR によって MBR テスターが異常終了した場合なので、評価値には最低の 0 を設定し、その個体は死んだ旨を出力します (feedback 関数で淘汰されます)。

個体数分の for ループを終えると、全個体の評価値の降順でソートした CSV(evaluation_value_list.csv) を生成し、evaluation 関数は終わりです。この CSV を次の feedbackn 関数で使います。

3.2.4 フィードバック

最後に feedback 関数の実装ですが、結構量があるので部分毎に分けて説明します。

feedback 関数では、まず、feedback 関数内で使用する定数を定義します (リスト 3.10)。

リスト 3.10 フィードバック (定数定義)

```
feedback() {
    local i

    echo 'feedback'
    echo '-----'

    candidates_num=$((POPULATION_SIZE / 2))
    mutation_num=$((POPULATION_SIZE / 5))
    top_num=$((POPULATION_SIZE - (candidates_num + mutation_num)))
```

"candidates_num"は交叉を行う候補となる個体の数です。全個体数の上位 50% で交叉を行い、同数の子供を次世代へ引き継ぎます。

"mutation_num"は突然変異させる個体の数です。交叉を行う候補とは別で選定し、突然変異させた後、次世代へ引き継ぎます。

"top_num"は何も変化を与えずにそのまま引継ぐ個体の数です。交叉をさせる 50% と突然変異させる 20% を除いた残り 30% をそのまま次世代へ引き継ぎます。30% の個体は評価値上位から選定します。

次に、evaluation 関数で評価値が 0 になった個体を淘汰します (リスト 3.11)。

リスト 3.11 フィードバック (淘汰)

```
# 淘汰
# evaluation_value=0 の個体を一番良い個体で置き換える
echo ">>>>>> To be culled"
top_ch=$(awk -F',' '{print $1;exit}' \
    ${WORK_DIR}/now/evaluation_value_list.csv)
echo "top_ch:${WORK_DIR}/now/${top_ch}"
for i in $(awk -F',' '$2==0{print $1}' \
    ${WORK_DIR}/now/evaluation_value_list.csv); do
    mkdir -p ${WORK_DIR}/${age}_culled
    mv ${WORK_DIR}/now/${i} ${WORK_DIR}/${age}_culled/
    echo "save $i to ${WORK_DIR}/${age}_culled/"
    cp ${WORK_DIR}/now/${top_ch} ${WORK_DIR}/now/${i}
    echo "replace ${WORK_DIR}/now/${i} with ${WORK_DIR}/now/${top_ch}"
done
echo
```

淘汰は、評価値が 0 であった個体について、now ディレクトリ内の当該個体を、一番評価値が高かった個体で置き換えることで行っています。

これだけで淘汰された MBR は上書きされて失われてしまうので、淘汰が発生する際は"<世代番号>_culled"というディレクトリを作成し、そこへ淘汰される MBR を事前にバックアップしています。

また、now ディレクトリ内のファイルだけ上書きして、CSV には反映させていません。これは交叉の際に候補となるようにしており、詳しくは後述します。

次に、評価値上位から全数の 30% の個体を次世代へ引き継ぎます (リスト 3.12)。

リスト 3.12 フィードバック (上位 30% を引継ぐ)

```
# (個体数 - (残した子孫の数 + 突然変異数)) 分の個体を、上位から順に next ヘコピー
echo ">>>>>> Copy Top ${top_num}"
for i in $(seq ${top_num}); do
    line=$(sed -n ${i}p ${WORK_DIR}/now/evaluation_value_list.csv)
    echo "line:$line"
    cp ${WORK_DIR}/now/${next}/${i} ${WORK_DIR}/now/${i}
    echo "save_to_next:${WORK_DIR}/now/${i}"
done
echo
```

top_num の数だけループを回し、評価値上位から順に next ディレクトリヘコピーしています。

続いて、全数の 50% で交叉を行います (リスト 3.13)。

リスト 3.13 フィードバック (交叉)

```
# 上位 50% の個体を候補に、同数の個体を next へ追加
# (一様交叉を使用)
echo '>>>>>>> Crossover'
max_cross_times=$((candidates_num / 2))
for cross_times in $(seq 0 $((max_cross_times - 1))); do
    echo "cross_times=${cross_times}"

    chA_idx=$((RANDOM % candidates_num) + 1)
    chB_idx=${chA_idx}
    while [ ${chB_idx} -eq ${chA_idx} ]; do
        chB_idx=$((RANDOM % candidates_num) + 1)
    done
    echo "chA_idx=${chA_idx} , chB_idx=${chB_idx}"

    chA=${WORK_DIR}/now/$(sed -n ${chA_idx}p \
        ${WORK_DIR}/now/evaluation_value_list.csv \
        | cut -d',' -f1)
    chB=${WORK_DIR}/now/$(sed -n ${chB_idx}p \
        ${WORK_DIR}/now/evaluation_value_list.csv \
        | cut -d',' -f1)
    echo "chA=${chA}"
    echo "chB=${chB}"
    partial_crossover $chA $chB \
        child$((2 * cross_times)).dat \
        child$((2 * cross_times + 1)).dat

    echo
done

cross_idx=0
for i in $(seq 0 $((POPULATION_SIZE - 1))); do
    [ ${cross_idx} -ge ${candidates_num} ] && break
    if [ ! -f ${WORK_DIR}/next/ch_${i}.dat ]; then
        mv child${cross_idx}.dat ${WORK_DIR}/next/ch_${i}.dat
        echo "mv child${cross_idx}.dat ${WORK_DIR}/next/ch_${i}.dat"
        cross_idx=$((cross_idx + 1))
    fi
done

echo
```

まず、1 つ目のループでは交叉の回数分 (候補数の半分) ループを回し、交叉対象の選択と交叉を行っています。変数 chA と chB を設定するところまでが交叉対象の選択です。交叉対象は評価結果の上位 50% の中から任意の 2 つを選択することで決めます。そして、交叉は partial_crossover という関数で行っています。

partial_crossover 関数 (部分交叉) はコードを見せるよりも図で示すと一目瞭然なので、図 3.1 に示します。単にランダムに選んだ 1 バイトを互いに入れ替えるだけです。

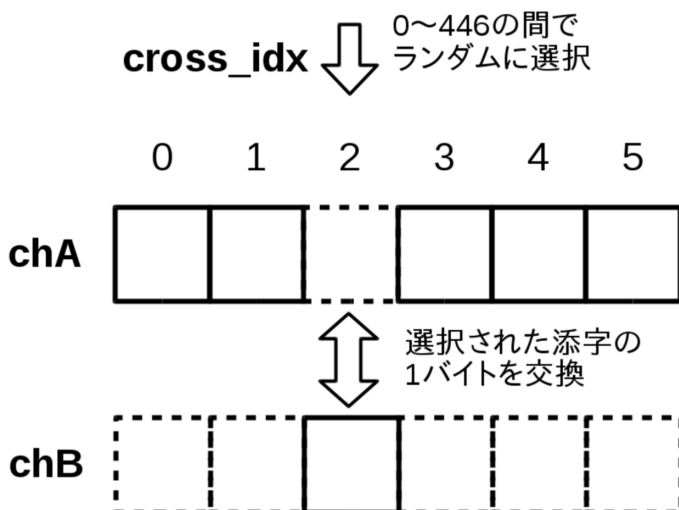


図 3.1 部分交叉について

なお、現在の実装では一度交叉を行った個体が再度選択される可能性があります。子供を残す個体も居れば、残さない個体も居る、という方が自然かなと思い、このようにしています*2。

そして、2 つ目のループでは 1 つ目のループで作った子供たちに番号を割り当て、next ディレクトリに"ch_番号.dat"というファイル名で保存しています。

最後に、残る 20% の個体を突然変異させます (リスト 3.14)。

リスト 3.14 フィードバック (突然変異)

```
# 個体数の 20% を突然変異させ、next へ追加
echo '>>>>>> Mutation'
for i in $(seq 0 $((POPULATION_SIZE - 1))); do
    if [ ! -f ${WORK_DIR}/next/ch_${i}.dat ]; then
        generate ${WORK_DIR}/next/ch_${i}.dat
        echo "mutated:${WORK_DIR}/next/ch_${i}.dat"
    fi
done
```

*2 と言いつつ、単に実装をサボっただけでもあります。

```
    echo  
    echo  
}
```

全個体番号の内、next ディレクトリにまだ居ない番号があれば、それを generate 関数で乱数列で生成し直しています。

部分交叉

部分交叉は、今回筆者の考えで導入しました。実際には遺伝的アルゴリズムの世界にそのような交叉方法があるのかわかりません。

当初は「数列の各値で入れ替えが発生するか否かを $1/2$ の確率で決める」という「一様交叉」で行っていました。しかし、世代を重ねても評価値が向上する様子が見られませんでした。

考えてみれば、機械語のバイナリ列は特定の 1 バイトを書き変えるだけでも動かなくなってしまうことさえあるので、現世代でせっかく良い評価であった個体も上位 30% に入れず、交差する運命となった時点で全く別物になってしまうことになります。

各世代で交叉する際の変更は 1 バイトだけにして、少しの変更とその結果を評価することを繰り返すようにしようと考え、1 バイトだけ入れ替える「部分交叉」を導入しました。

もっと考えてみると、実行バイナリと言えど、それは機械語という言語の CPU が解釈するソースコードと言えなくもないものです。普段のプログラミングで、(行き詰まったときなどは特に、)「少し書き換えて実行結果をみて、また少し書き変える」ということを繰り返していたりします。それを考えると、普段のプログラミングという行為を遺伝的アルゴリズムで表現するならば「部分交叉」という交叉方法になるのかな、と思います。

3.3 動作確認

3.3.1 テストを用意

MBR テスターの `test_ch` 関数をすこし変更します。「"A"という文字を出力できたら pass」ではあまり面白くないので、「ASCII の表示可能文字を出力できたら pass」というようにテストを変更し、遺伝的アルゴリズムではじめて出力する文字は何かを試してみます。

`test_ch` 関数をリスト 3.15 のように変更します。

リスト 3.15 ASCII 表示可能文字を出力するかテスト

```
static uint8 test_ch(uint8 ch)
{
    if ((0x21 <= ch) && (ch <= 0x7e)) {
        printf("ch=%c(0x%02x)\n", ch, ch);
        return 1;
    } else
        return 0;
}
```

変更後、コンパイルし、`mbr_tester` バイナリを `ge_mbr.sh` があるディレクトリへ配置してください。また、`mbr_tester` は `seabios.bin` と `vgabios.bin` も必要とするため、同じディレクトリへ配置してください。

3.3.2 実行結果

実行すると、図 3.2 のようにログが出力されていきます。

```
→ ge_mbr_v1.0 ./ge_mbr.sh
Initialization
initialization
-----
WORK_DIR=20180916224253
20180916224253/now/ch_0.dat
446+0 レコード入力
446+0 レコード出力
446 bytes copied, 0.00133363 s, 334 kB/s
20180916224253/now/ch_1.dat
446+0 レコード入力
446+0 レコード出力
446 bytes copied, 0.00137693 s, 324 kB/s
20180916224253/now/ch_2.dat
446+0 レコード入力
446+0 レコード出力
446 bytes copied, 0.00154844 s, 288 kB/s
20180916224253/now/ch_3.dat
446+0 レコード入力
446+0 レコード出力
```

図 3.2 ge_mbr.sh の実行ログ

待っていると、何やら文字を出力したようです！ その文字は"B"みたいですね (図 3.3)。

```
i=14
>>>> 20180916224253/now/ch_14.dat
ch=B(0x42)
pass(test_counter=157)
20180916224253/now/ch_14.dat has reached the answer!
evaluation_value=100
```

図 3.3 初めての文字は"B"

何を出力させていたのか、動作させて見てみます。前章にて、GUI に表示させる内容をコンソールに出力させるように改造した `kvmulate` で実行してみると、図 3.4 のように

なっていました。

```
-----
00: SeaBIOS (version 1.7.5-20140531_083029-gandalf)
01:
02: Press F12 for boot menu.
03:
04: Booting from Floppy...
05: Booting from Hard Disk...
06: Boot failed: could not read the boot disk
07:
08: No bootable device.  Retrying in 60 seconds.
09:
10:
11:
12:
13:
14:
15:
16:
17:
18:
19:
20:
21:
22:
23:
24:
-----
```

図 3.4 "B"が出力されていた正体

MBR が出力していたのではなく、MBR はどうにかしてフロッピーディスクからのブートを失敗させて、BIOS に HDD からのブートを試させていたのです。

「MBR に何らかの文字を出力するよう進化してほしい」という思惑からは外れましたが、「フロッピーディスクからブートする旨のメッセージが出力される 4 行目以降に何らかのメッセージを出力する」というテスト要件は満たしています。「ずるい手を使われたな」と思いつつも、神様の思ったとおりには動いてくれないものだな、となんだか面白い結果でした。

3.4 補足: 評価方法を工夫する (MBRatoon)

やってみると分かりますが、テストを適切に用意してあげないと、今回の様にちょっとずるい手に走ってしまったりします (それはそれで面白いのですが)。

そもそも 0 点が 100 点しかない評価方法だと、100 点の個体が生まれるまで全員 0 点ということになり、「優れたものを残し、掛け合わせることで答えに近づける」という流れにできません。やはり評価としては pass/fail ではなく、点数を返すような評価方法じゃないといつまでたっても成長しないですね。(当たり前のことかもしれませんが。)

そこで、数値的に評価できる指標として、「画面を描画したピクセル数」を評価値としてみます。

MBR テスターと `ge_mbr.sh` を変更します。変更箇所の要点だけ説明すると、以下の通りです。

- MBR テスター
 - SDL を消す前に戻す
 - `viewer.c` の `updateTextmode` 関数で、ある一定回数以上画面描画を行ったとき、何らかの色で描画されている (0 でない) ピクセル数を標準出力に出して終了する
 - 画面のピクセル情報は `pScreen->pixels` から参照できる
 - ピクセル値の参照方法は `drawGlyphAt` 関数などの実装を参照
- `ge_mbr.sh`
 - `evaluation` 関数で、MBR テスターの終了ステータスが 0 の時は `evaluation_value` 変数に MBR テスターが出力する評価値を格納するようにする

詳細は、MBR テスターの差分については筆者のウェブページを^{*3}、`ge_mbr.sh` はリポジトリの `test_fill` ブランチ^{*4}を見てみてください。

紹介したいのはこの結果です。

`ge_mbr.sh` 実行後、最初のころは、BIOS が出力するメッセージ分のピクセル数 (5,000pixel 程度) がほとんどの個体の評価値でした。

628 世代で評価値が急に 13,000pixel へ上がりました。1 つ目の進化です。これは、BIOS が大量のエラーログを吐き出しているためで、パターンとしては前述の「B」を出力した時と同じです。そしてこれを超える進化は一向に起こらず、この個体の繁栄が続きます。

^{*3} <http://yuma.ohgami.jp>

^{*4} https://github.com/cupnes/ge_mbr/tree/test_fill

驚くべきは 2510 世代で、この世代の 19 番目の個体 (ch_19.dat) の評価をする際、MBR テスターが終了せずに固まっていました。画面をよく見ると、文字ではなくカラフルなドットが描画されていました。何が起きたのかというと、ASCII を VRAM へ書き込んで文字を出力する「テキストモード」から、VRAM へピクセルフォーマットに従ったピクセルデータを書き込んで描画する「グラフィックモード」へビデオモードを変更していたのでした。「一定時間待って描画ピクセル数を数え、終了する」処理は「updateTextmode 関数」側に入れていたため、グラフィックモードに移行すると VM である MBR テスターが終了せず、そのまま固まっていたのです。

これは偉大な進化です。しかし、この時は固まった MBR テスターを kill で終了させるしかなく、その場合、この個体の評価値は 0 となるため、次の世代で淘汰されたのか、居なくなっていました。ただ、過去の個体データはすべて残っているため、過去の個体データから実行を再開し、同じ状況を再現したのが表紙の写真です*⁵。

*⁵ なお、表紙は「2001 年宇宙の旅」が 2001 年にアメリカで再上映された際のポスターが元ネタです。元ネタのポスターでは、MBR を実行しているウィンドウのあたりにスターチャイルドが居て、こっちを見えています。それにならって、進化した MBR がこっちを見ているイメージです。

おわりに

本書をお読みいただきありがとうございました！

本書では `/dev/kvm` を直接叩くところから、KVM を扱う既存のコードの改造して MBR のテスターを作成し、遺伝的アルゴリズムで遺伝的 MBR を実現しました。

そもそも、自作 OS 自動化を考えるとときに遺伝的アルゴリズムが適切なのかという議論もあります。言い訳というわけではない、いや、言い訳ですが、今回は「フルスクラッチで作る!」シリーズの進捗が悪く、サブプロジェクト的にちょこちょこ進めていたことで何か本を出そうと切り替えました。`/dev/kvm` を直接叩く、というネタを少し前に自作 OS をやる裏で進めていて、遺伝的アルゴリズムをシェルスクリプトで書く、というネタは自作 OS を始める前にやっていたものです (どこにも公開していないので HDD に眠っていたものですが)。

そんな「冷蔵庫の余り物で一品」的にはなんだかまとまった (?) 上に、やはり最終的にはこれも自作 OS になりました。

自作 OS に限らず、プログラミングというものが目的のバイナリを得るための手法なのだとしたら、それは必ずしも人間がエディタとコンパイラを駆使して行う必要は無いのではないかと思います。普段私達がやっているプログラミングが、「1) たたき台を作る」・「2) 実行し結果を評価」・「3) 評価を踏まえて改善」・「以降は 2) と 3) の繰り返し」、という形でモデル化できるのならば、たたき台は乱数列で用意するか近しい挙動をする既存のものを使って、後は評価と改善の方法を自動化できればプログラミングという行為を全て自動化できることになります。

その意味では、一応、本書はこの 3 ステップを全て自動化できており、PoC としては良いんじゃないかなと、思います。

これがもっと上手く行くようになれば、自作 OS も自分が頑張る他に自動化スクリプトが頑張ってくれる複数プロジェクト同時並列が可能に。。? なるといいなあ

参考情報

参考にさせてもらった情報

- The Definitive KVM (Kernel-based Virtual Machine) API Documentation
 - <https://www.kernel.org/doc/Documentation/virtual/kvm/api.txt>
 - Linux カーネル付属のドキュメント
 - KVM の API に関する一次情報
- kvmulate
 - <https://github.com/slowcoder/kvmulate>
 - /dev/kvm を直接叩くというのはそもそも情報が無く、こういった GitHub 上で公開されているソースコードから勉強させていただきました。大変ありがたいです。
- 遺伝的アルゴリズム - Wikipedia
 - <https://ja.wikipedia.org/wiki/遺伝的アルゴリズム>

自作 OS 自動化の PoC としての遺伝的 MBR

2018 年 10 月 8 日 技術書典 5 版 v1.0

著 者 大神祐真

発行所 へにゃぺんて

連絡先 yuma@ohgami.jp

印刷所 日光企画

(C) 2018 へにゃぺんて