毛利研究室輪講資料　令和 4 年度 第 3 回　令和 4 年 10 月 28 日（金）

xv6-publicにおけるブートローダの調査

山下 恭平

概要：xv6-public[1] は MIT が開発した x86 アーキテクチャで動作する教育用オペレーティングシステム (OS) である.xv6 を動作させるにはプロセッサエミュレータ QEMU を使用する. 本稿では QEMU に xv6 を起動するように命令をしてから,xv6 のカーネルがメモリ上に配置されるまでの一連の動作について調査を行い,BIOS とブートドライブの関係を明らかにし、ブートローダがカーネルをメモリにロードするプロセスを明らかにする.

1. はじめに

コンピュータが起動してからオペレーティングシステム (OS) が動作するまでの間に, 様々なプログラム (BIOS やブートローダ) が動作している. しかし, 私は OS が動作する前に動作するプログラムについての知見を持ち合わせていない. そこで, 教育用 OS の xv6-public を用いてブートローダについて調査を行い, カーネルがメモリ上にロードされるまでの動作について明らかにする

本稿では第 2 章で実験環境について説明し, 第 3 章では xv6 の起動手順を示し BIOS とブートドライブの関係を確認する, 第 4 章では実際にブートドライブが作成される手順を確認し, 第 5 章でブートローダの動作を明らかにし, 第 6 章でそれらをまとめる.

2. 動作環境

用いた実験環境について表 1 と図 1 に示す.

表 1　動作環境

用途	名称
ホスト OS	macOS Monterey ver12.6
CPU	Intel Core i5-8279U
プロセッサエミュレータ	qemu-system-i386
ゲスト OS	xv6-public

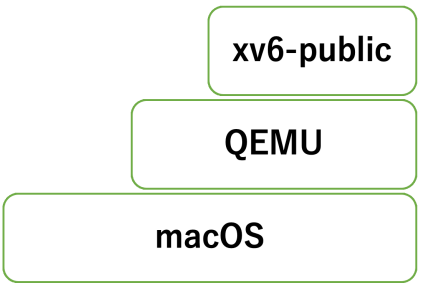

図 1　動作環境概略図

3. xv6-public の起動手順

調査対象を明確にするために xv6-public の起動手順を示し, ブートドライブと BIOS の関係を確認する

「make qemu」コマンド実行から xv6-public のカーネルがロードされるまでの手順は次の通りである

1). BIOS が起動する
2). qemu のオプションで指定されているブートドライブの MBR 内ブートシグネチャを確認
3). ブートシグニチャにマジックナンバー aa55 が格納されているため, ブートセクタと判断し MBR をメモリの 0x7c00 番地へロード
4). 0x7c00 番地にロードされたブートローダが動き始める
5). ブートローダによってカーネルがロードされる

ここで, ブートドライブの先頭 1 セクタ (512byte) のことを MBR(Master Boot Record) と呼び, さらに, その末尾 2byte をブートシグニチャという. ブートシグニチャにマッジクナンバー「aa55」が格納されているときに限り BIOS は指定したディスク (今回は xv6.img というファイル) をブートドライブとして認識する仕組みが存在する. そのため,xv6.img を設計するにあたって, 組み込まなければならない条件が存在することがわかった. 以下はその条件である

1). xv6.img の先頭 512byte にブートローダを配置する
2). ブートシグニチャにマジックナンバーを格納する

次項ではこの条件に合致する xv6.img の作成手法について確認する

4. ブートドライブの作成

前項で示した条件に合致するように xv6.img を作成しなければ xv6-public は正しく起動しない,xv6-public では「make」コマンドを実行すると自動的に xv6.img が生成される. ここで,Makefile を調査することで xv6.img の作成プロセスを明らかにしブートドライブの適切な設計手法につ

いて学ぶこととした. 図 2 に xv6.img を作るのに使用されている Makefile 内のコマンドを示す.

```
xv6.img: bootblock kernel
    dd if=/dev/zero of=xv6.img count=10000
    dd if=bootblock of=xv6.img conv=notrunc
    dd if=kernel of=xv6.img seek=1 conv=notrunc
```

図 2 xv6.img の Makefile

Makefile 内のコマンドから xv6.img についてわかることを以下にまとめる.

1). dd コマンドによって 10000 ブロックの空白のファイルが生成される
2). 先頭の 1 ブロックに bootblock を書き込む
3). 2 ブロック目に kernel を書き込む

ここで dd コマンドとは, ブロック単位でデータを読み書きするコマンドであり,1 ブロックはデフォルトで 512byte となっている. また,seek オプションでは書き込み開始にあたってスキップするブロック数を指定する, ここでは kernel の書き込みに際して「seek = 1」が指定されていることから,kernel は xv6.img の先頭 1 ブロックをスキップした 513byte 目から書き込まれていることがわかる. 図 4 に xv6.img の構成イメージを示す.

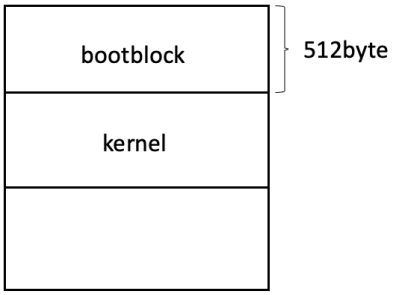

図 3 xv6.img の構成イメージ

前項で説明したように,xv6.img の先頭 512byte にはブートローダが格納されている, 従って,bootblock がブートローダ本体である. 次に,bootblock を詳しく確認していく.

図 4 に bootblock を生成する Makefile 内のコマンドを示す.

```
bootblock: bootasm.S bootmain.c
    $(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c bootmain.c
    $(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S
    $(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o bootmain.o
    $(OBJDUMP) -S bootblock.o > bootblock.asm
    $(OBJCOPY) -S -O binary -j .text bootblock.o bootblock
    ./sign.pl bootblock
```

図 4 bootblock の Makefile

bootblock は bootasm.S と bootmain.c の二つから構成されていることが確認できる.bootasm.S と bootmain.c をそれぞれコンパイルし, 生成されたオブジェクトファイルを ld コマンドでリンクすることで bootblock.o を生成している. このとき「-Ttext」オプションにより開始アドレスを 0x7C00 番地に設定しているのがわかる, これは BIOS がブートローダを 0x7C00 番地へ格納することからこの地点を開始アドレスに設定していると考えられる. ここで生成された bootblock.o を objcopy コマンドで変換し bootblock が生成される. 図 5 に bootblock の構成イメージを示す.

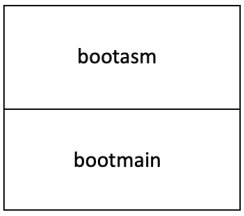

図 5 bootblock の構成イメージ

最後に, 生成された bootblock を sign.pl というプログラムに読み込ませていることがわかる. そこで,sign.pl のソースコード (図 6) を確認すると, これは bootblock の末尾にブートシグニチャを書き込むプログラムだということが確認できた.sign.pl によって xv6.img の先頭 512byte に格納されるブートローダが完成する.

```
open(SIG, $ARGV[0]) || die "open $ARGV[0]: $!";

$n = sysread(SIG, $buf, 1000);

if($n > 510){
  print STDERR "boot block too large: $n bytes (max 510)\n";
  exit 1;
}

print STDERR "boot block is $n bytes (max 510)\n";

$buf .= "\0" x (510-$n);
$buf .= "\x55\xAA";

open(SIG, ">$ARGV[0]") || die "open >$ARGV[0]: $!";
print SIG $buf;
close SIG;
```

図 6 sign.pl のソースコード

ここまでの一連の流れにより, 第 3 章で示した条件を満たす xv6.img が生成されることが明らかとなった. 次の章からは実際のブートローダの動作を確認していく.

5. ブートローダの動作

xv6-public におけるブートローダは xv6.img に格納されている bootblock であることがわかった。この章ではブートローダの動作の詳細を明らかにすることを目的とする。bootblock は bootasm と bootmain の 2 つのプログラムで構成されている, そのそれぞれの詳細な動作について調査を行ったが, 私の理解が追いつかない部分が非常に多かったため, それぞれの動作の概略についてまとめる.

5.1 bootasm

bootasm は以下のことを行っていた

1). 各種レジスタの初期化
- 図 7 に bootasm のアセンブリの一部を示す
- %ax レジスタを 0 に初期化しそれを他のレジスタにコピーしている

2). A20Line の有効化
- x86 ベースコンピュータのシステムバスを構成する電気線の 1 つ

3). GDT の設定
- Global Descriptor Table のこと
- セグメントディスクリプタのセグメント管理に使用

4). プロテクトモードへの移行
- x86 アーキテクチャの動作モードの一つ
- 初めはリアルモード (16bit) で動くがプロテクトモードへ移行することで 32bit で動作するようになる
- CR0 レジスタ (図 8[1]) の 0bit 目を 1 にすることで移行できる

5). bootmain の呼び出し

```
10  .code16                     # Assemble for 16-bit mode
11  .globl start
12  start:
13    cli                       # BIOS enabled interrupts; disable
14
15    # Zero data segment registers DS, ES, and SS.
16    xorw    %ax,%ax           # Set %ax to zero
17    movw    %ax,%ds           # -> Data Segment
18    movw    %ax,%es           # -> Extra Segment
19    movw    %ax,%ss           # -> Stack Segment
20
```

図 7 各レジスタの初期化

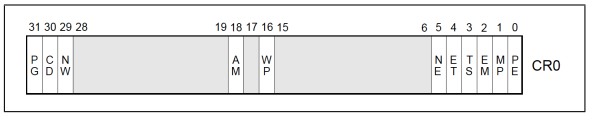

図 8 コントロールレジスタ 0

5.2 bootmain

1). xv6.img の 2 セクタ目をロード
- ソースコード 28 行目 (図 9)

2). elf ヘッダを持つか確認
- ソースコード 31 行目

3). program ヘッダから情報を取得しカーネルをロード
- ソースコード 35 行目から 42 行目

4). entry() の呼び出し
- ソースコード 47 行目

bootmain はカーネルをメモリ上にロードすることを目的としたプログラムである．28 行目で取得したプログラムヘッダの情報をもとに 39 行目で readseg() 関数を呼び出すことで, カーネルをメモリにロードしている. ここで readseg 関数は第 1 引数にロード先, 第 2 引数にロードするデータサイズ, 第 3 引数に xv6.img の 2 ブロック目からのオフセットを入力する. つまり,28 行目の readseg では変数 elf に,xv6.img の 2 セクタ目の先頭から,4096byte 分の情報をロードしている. このとき xv6.img の 2 セクタ目には kernel が格納されている. 図 10 はプログラムヘッダの内容を objdump コマンドにより表示させた物である. ここに格納されている情報が 39 行目の readseg() の引数として使用されることで, メモリ上にカーネルがロードされる.

```
17  void
18  bootmain(void)
19  {
20    struct elfhdr *elf;
21    struct proghdr *ph, *eph;
22    void (*entry)(void);
23    uchar* pa;
24
25    elf = (struct elfhdr*)0x10000;  // scratch space
26
27    // Read 1st page off disk
28    readseg((uchar*)elf, 4096, 0);
29
30    // Is this an ELF executable?
31    if(elf->magic != ELF_MAGIC)
32      return;  // let bootasm.S handle error
33
34    // Load each program segment (ignores ph flags).
35    ph = (struct proghdr*)((uchar*)elf + elf->phoff);
36    eph = ph + elf->phnum;
37    for(; ph < eph; ph++){
38      pa = (uchar*)ph->paddr;
39      readseg(pa, ph->filesz, ph->off);
40      if(ph->memsz > ph->filesz)
41        stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
42    }
43
44    // Call the entry point from the ELF header.
45    // Does not return!
46    entry = (void(*)(void))(elf->entry);
47    entry();
48  }
```

図 9 bootmain.c

```
kyoheiyamashita@Kyohei-Yamashita-MacBook-Pro-4 xv6-public % objdump -p kernel

kernel: file format elf32-i386

Program Header:
    LOAD off    0x00001000 vaddr 0x80100000 paddr 0x00100000 align 2**12
         filesz 0x00006eac memsz 0x00006eac flags r-x
    LOAD off    0x00008000 vaddr 0x80107000 paddr 0x00107000 align 2**12
         filesz 0x00002516 memsz 0x0000d4a8 flags rw-
```

図 10 プログラムヘッダの情報

6. おわりに

xv6-public が起動してからカーネルがロードされるまでの動作について, ブートドライブの構成から明らかにすることで理解を深めることができた. ブートドライブとブートローダの設計には,OS についての知識だけでなく BIOS や x86 アーキテクチャなど低階層における知識を体系的に取得している必要があることを知ることができた. そのため, 本稿の後半部分に当たる, 実際にブートローダの動作を確認を行う調査が私の知識不足より, 浅い理解しかできず, 資料の作成にとても戸惑ってしまった. 今後は, 私自身の知識の補完を行ってから再び調査を行うことや,BIOS ではなく UEFI についての調査も行っていきたい.

参考文献

[1] mit-pods/xv6-public <https://github.com/mit-pdos/xv6-public >(accessed 2022-10-30).

[2] Intel 64 and IA-32 Architectures Software Developer’ s Manual, Volume 3A: System Programming Guide, Part 1, P74 <https://www.intel.co.jp/content/www/jp/ja/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html >(accessed 2022-10-30).