フルスクラッチで作る!
x86_64 自作 OS

大神祐真　著

2018-04-22 版　　へにゃぺんて 発行

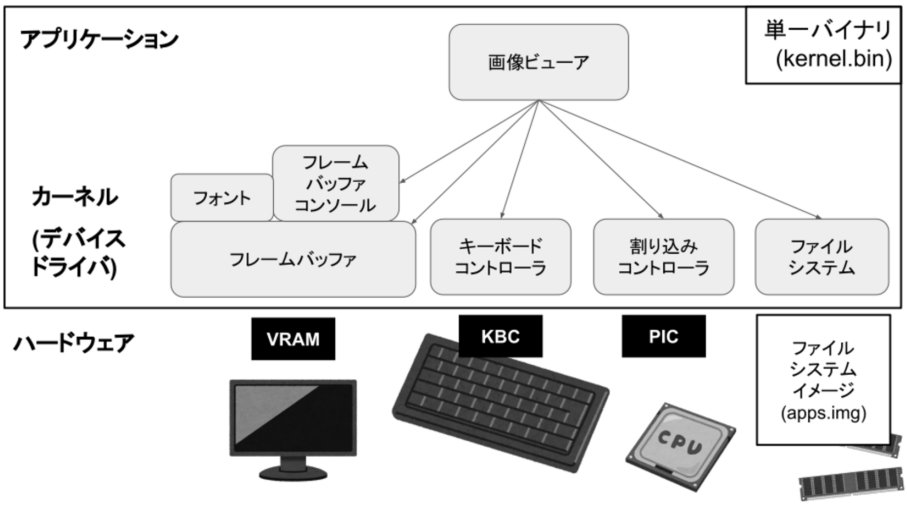
はじめに

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 64 ビットの x86 CPU(x86_64) を対象にフルスクラッチで"OS"を作ってみる本です。本書では主に"カーネル"と呼ばれるハードウェアの機能を上位のアプリケーションに向けて抽象化する部分を作ってみます。

そして、上記を使用したアプリケーション (的なもの) として、本書の最後に「画像ビューア」を作ってみます。本書ではカーネルとアプリのバイナリレベルでの分離などは行わないため、アプリケーションはカーネルのバイナリに含まれたものとなります (そのため、「アプリケーション (的なもの)」です)。

本書で作成するカーネルとアプリケーションの構成は図 .1 の通りです。



▲図 1　本書で作成するソフトウェア構成

動作確認しているハードウェアとブートローダーについて

本書ではカーネルとそれより上位のソフトウェアレイヤーをすべてフルスクラッチで自作します。

カーネルを起動させるためのブートローダーと、それらを動作させるハードウェア環境は筆者は以下で動作確認しています。

- ハードウェア
  - CPU エミュレータ: QEMU
  - 実機: Lenovo 製 Thinkpad E450
- ブートローダー
  - poiboot

ハードウェアについて、本書の内容を試す際は、まずは QEMU で試してみることをおすすめします。実機での動作確認は筆者の所持している 64 ビット PC の都合上、Lenovo 製 Thinkpad E450 のみです。実機で試す際におそらく問題となるのは 3 章で、いわゆる「PS/2 マウス・キーボード」を制御するための IC@<intro_jissai_kbc>であるキーボードコントローラ (KBC) を使用します。ノート PC の場合、内蔵されているキーボードが内部的には KBC に接続されていたりしますが、デスクトップ PC の場合、PS/2 コネクタが搭載されていない場合はそもそも KBC も搭載されていないかと思われます[*1]。

ブートローダーは"poiboot"というものを使用します。実はこれは本書のシリーズである「フルスクラッチで作る!UEFI ベアメタルプログラミング」のパート 1(無印) とパート 2 で説明した内容に少し手を加えて作ったものです。ブートローダーは本書の範囲を超えるため、手を加えた内容については本書の「付録」で簡単に説明するにとどめています。

開発環境について

本書で行う開発は、以下の環境で行うものとして説明します。

- CPU: x86_64
- OS: Debian GNU/Linux 8.0(Jessie)

[*1] KBC と言っても今や CPU(SoC) 集積されている「KBC 互換の何か」なので、もしかすると PS/2 コネクタが無い様なマザーボードには USB 対応の KBC 互換の何かが搭載されているのかも知れませんが試したことも無く、わかりません。

基本的に使用するツールはエディタ (お好きなものを)、コンパイラ (GCC)、make 程度なので、それらのツールの Windows 版を使用する等すれば他の OS でも開発を行うことができるかと思います。

ただし、5 章でシェルスクリプトを作成する箇所や、6 章で"ImageMagick"という画像処理系のツールセットを使用する箇所があります。Linux 以外の OS で開発する場合は適宜読み替えや、Windows 版の導入等を行ってください。

本書の PDF 版やサンプル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 版は以下のページで無料で公開しています。

- `http://yuma.ohgami.jp`

また、本書で扱うサンプルコードとコンパイル済みバイナリは以下の GitHub で公開しています。

- ソースコード
  - `https://github.com/cupnes/x86_64_jisaku_os_samples`
- コンパイル済みバイナリ
  - `https://github.com/cupnes/x86_64_jisaku_os_samples/releases`

サンプルコードは節ごとにディレクトリが分かれています。本書の各節冒頭で「この節のサンプルコードは"〜"のディレクトリです」のように説明していますので、適宜参照してください。

目次

はじめに **2**

動作確認しているハードウェアとブートローダーについて 3

開発環境について 3

本書の PDF 版やサンプルコードの公開場所について 4

第 1 章　開発環境とブートローダーの準備 **9**

1.1　開発環境の準備 9

1.2　ブートローダー (poiboot) の準備 10

1.2.1　ブートローダーの役割 10

1.2.2　本書で使用するブートローダー"poiboot"について 11

1.2.3　poiboot をダウンロード 11

1.2.4　設定ファイル (poiboot.conf) について 12

1.2.5　メモリマップを見てみる 12

1.2.6　メモリマップの内容から poiboot.conf 作成 15

第 2 章　フレームバッファを制御する **17**

2.1　画面を単色で塗りつぶす 18

2.2　ソースコードを書く 18

2.3　リンカスクリプトを書く 19

2.4　Makefile を書く 21

2.5　コンパイルする 22

2.6　QEMU で実行する 22

2.7　実機で試す 25

2.7.1　USB フラッシュメモリへファイル配置・起動 25

2.8　ソースコードを整理 26

2.9　画面に文字を表示する (フォント) 29

第 3 章　キーボード入力　37
3.1　キーボードコントローラ (KBC) の使い方 38
3.2　ステータスレジスタとデータレジスタ 38
3.3　インラインアセンブラ 39
3.4　キー入力を取得する 41

第 4 章　割り込みを使う　46
4.1　CPU: セグメンテーション設定 47
4.1.1　GDT の定義 47
4.1.2　レジスタ GDTR の設定 (lgdt 命令) 48
4.1.3　レジスタ CS・DS・SS の設定 48
4.1.4　まとめ: gdt_init 関数の定義 49
4.2　CPU: IDT 設定 50
4.2.1　ハンドラ作成 51
4.2.2　IDT の定義 55
4.2.3　レジスタ IDTR の設定 (lidt 命令) 57
4.3　CPU: 割り込み有効化 (sti 命令) 58
4.4　PIC: 割り込みコントローラ (PIC) 設定 59
4.4.1　PIC 初期化処理の流れと使用するレジスタについて 59
4.4.2　PIC 初期化処理の実装 61
4.4.3　割り込み有効化関数の実装 62
4.4.4　PIC の EOI(End Of Interrupt) 設定 64
4.5　仕上げ: main.c と Makefile へ反映する 67
4.5.1　main.c と Makefile へこれまでの変更を盛り込む 67
4.5.2　halt で待つようにする 68

第 5 章　メモリ上の簡易ファイルシステムを用意する　70
5.1　poiboot の第 2 のロード機能 71
5.2　単一のテキストファイルを読んでみる 71
5.3　ファイルシステムを考える 73
5.4　ファイルシステムを実装する 74
5.5　ファイルシステムイメージ (fs.img) 出力スクリプトを作る 78

第 6 章　画像ビューアアプリを作ってみる　80
6.1　仕様 81
6.2　画像ファイルを用意する 81

6.2.1 putd 関数を追加し、フレームバッファ解像度を画面表示して確認する 81
6.2.2 ImageMagick(convert コマンド) で画像を変換する 84
6.3 1 枚の画像を表示してみる 85
6.4 ファイルシステムからファイルの一覧情報を取得する 88
6.5 画像ビューアを実装する 90
6.5.1 実装方針 90
6.5.2 最初の画像表示処理を実装する (iv_init) 90
6.5.3 KBC 割り込み時の処理を実装する (iv_kbc_handler) 93

付録 A 表紙について 97
A.1 フォントジェネレータ (create_font.sh) 98

付録 B poiboot について 99
B.1 poiboot の構成 99
B.2 前著の + α (ExitBootServices) 101
B.3 UEFI のファイル読み出しが正しく動作しないケースがありました . . . 102

付録 C GPT フォーマット方法 104

おわりに 106

参考情報 107
参考にさせてもらった情報 107
本シリーズの過去の著作 107

第 1 章

開発環境とブートローダーの準備

1.1 開発環境の準備

本書では C 言語と一部アセンブラを使用して開発を行います。そのため、最低限、「コンパイラ/アセンブラ (GCC)」と「エディタ (好きなものを)」があれば良いです。加えて、作成したバイナリを試すための「エミュレータ (QEMU)」や、ビルドルールを管理するために「make」等を適宜インストールします。

筆者の作業環境の都合上、開発環境構築は Debian(バージョン 8 Jessie) で説明しますが、同じくパッケージ管理に APT を使用する Ubuntu でも同様に構築できると思います。

インストールするパッケージは以下の通りです。

- build-essential
  - GCC や make 等が含まれるパッケージ
- qemu-system-x86
  - 64 ビットの x86CPU のエミュレータ
- ovmf
  - 仮想マシン向け UEFI ファームウェア (後述)

以下のコマンドでインストールできます。

```
$ sudo apt install build-essential qemu-system-x86 ovmf
```

Ubuntu には ovmf パッケージが無い様です。結局は"OVMF.fd"というファームウェアバイナリが欲しいだけなので、ovmf パッケージが無ければ以下からバイナリをダウンロードしてください。

- `https://sourceforge.net/projects/edk2/files/OVMF/`

– 2018/01 現在、"OVMF-X64-r15214.zip"が OVMF.fd を含む最新のアーカイブです

■メモ: OVMF(Open Virtual Machine Firmware) について

"ovmf"パッケージは QEMU に含まれていない UEFI ファームウェアを追加するためにインストールしています。最近の PC ではレガシー BIOS から UEFI へ BIOS のファームウェア実装が変わっているのですが、QEMU にはレガシー BIOS しか含まれていないため、"OVMF(Open Virtual Machine Firmware)"という仮想マシン向けの UEFI 仕様準拠のファームウェアバイナリをインストールしています。

UEFI については Google 検索でもいろいろと情報が出てきます。本シリーズの前著でも「UEFI ファームウェアを直接叩くプログラムを作る」本を出しており、PDF 版は無料で公開していますので興味があれば見てもらえると嬉しいです (本書の「参考情報」に URL を記載しています)。

1.2 ブートローダー (poiboot) の準備

1.2.1 ブートローダーの役割

ブートローダーについて少し説明します。

PC は電源を入れるとレガシー BIOS や UEFI といったファームウェアが動作します。ファームウェアは、レガシー BIOS であれば起動ディスクの先頭 512 バイト (マスタブートレコード) を、UEFI であれば起動ディスク第 1 パーティションの特定のパスにある実行バイナリをメモリへロードし実行します。

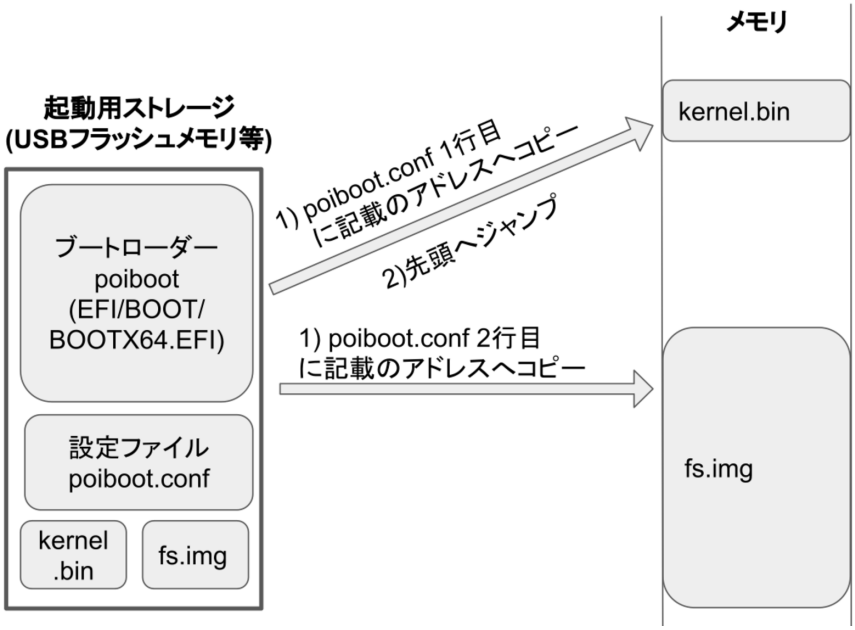
ここで実行されるプログラムが一般的にブートローダーと呼ばれるものです。ブートローダーはファームウェアとやり取りして、その次に実行するプログラム (一般的には「カーネル」) をメモリへロードし、ファームウェアとのやり取りを終え[*1]、メモリへロードしたプログラムの実行を開始します。

[*1] ファームウェア動作中はファームウェアが各種ハードウェアを管理しており、特定のメモリ領域が使えなかったりします。カーネル起動後はカーネル側でこれらのリソースを管理したいため、ファームウェアが動作できないモードへ移行します。

1.2.2 本書で使用するブートローダー"poiboot"について

本書では前著までの「フルスクラッチで作る!UEFI ベアメタルプログラミング」シリーズ[*2]で紹介した内容とプラスアルファ (付録で簡単に紹介しています) の内容を元に作ったブートローダー"poiboot"を使用します。

poiboot の動作は図 1.1 の通りです。



▲図 1.1　poiboot の動作

poiboot は設定ファイル (poiboot.conf) に記載されているアドレスへカーネルバイナリ (kernel.bin) とファイルシステムイメージ (fs.img) をロードします。

1.2.3 poiboot をダウンロード

それでは、poiboot を準備していきます。

まず、以下の URL から poiboot のアーカイブ"poiboot_20180422.zip"をダウンロード

[*2] PDF 版は無料で公開していますので興味のある方は「参考情報」の章に記載の URL からダウンロードして見てみてください。

し、展開しておいてください。

- `http://yuma.ohgami.jp`

展開すると中には"poiboot.efi"と"poiboot.conf"が入っています。poiboot.efi がブートローダーの実行バイナリです。

poiboot.efi を何処に配置して使用するかは次章で最初のプログラムを作っていく中で紹介します。

1.2.4 設定ファイル (poiboot.conf) について

poiboot 準備の最後として、設定ファイル (poiboot.conf) を用意します。設定内容は、poiboot がロードする kernel.bin と fs.img それぞれのロード先アドレスだけです。

なお、poiboot のアーカイブに入っているデフォルト値記載済の poiboot.conf をまずは使用してみて、ダメだったらメモリマップ確認 (後述) を行う、でも良いかと思います[*3]。QEMU は仕様が変わらない限りメモリマップは変わらないと思いますし、実機の場合もネット上で実機のメモリマップをいくつか見る限り[*4]、デフォルトの poiboot.conf で問題なさそうです。

1.2.5 メモリマップを見てみる

それでは、自作 OS を実行する環境のメモリマップ[*5]を見てみましょう。

メモリマップを見るのに本書では「UEFI シェル」を使います。UEFI シェルは UEFI ファームウェアが持つ機能へアクセスする為のシェルです。このシェル上で"memmap"というコマンドを実行してメモリマップを確認します。

UEFI シェルを起動する

まずは UEFI シェルを起動します。「QEMU の場合」・「実機の場合」それぞれで紹介します。

■ QEMU の場合

[*3] ただ、メモリマップ確認には UEFI シェルというものを使うのですが、自身の PC のハードウェア情報を知る便利なツールなので興味があればぜひ試してみて欲しいです。

[*4] "uefi memmap"等で画像検索すると、実機で memmap コマンドを実行した結果の写真がいくつかヒットします。

[*5] 厳密には、ここでは「物理アドレス」に対応したメモリマップを確認します。CPU は MMU(メモリ管理ユニット) により変換された「仮想アドレス」で動作し、一般的なカーネルは仮想アドレスのメモリマップを何パターンも持っています。ただし UEFI の場合、ファームウェア側で「物理アドレス=仮想アドレス」となるように MMU の設定を行ってくれます。本書では「物理アドレス=仮想アドレス」設定のまま使用するため特に仮想アドレスのメモリマップは意識しません。

QEMU で使用する UEFI ファームウェア"OVMF"には UEFI シェルが内蔵されています。そのため、QEMU に"-bios"オプションで OVMF ファームウェアを指定して起動し、しばらく待っていると"Shell> "というプロンプトが表示されます。

起動するコマンドとしては以下の通りです。

```
■ APT で ovmf パッケージをインストールした場合
$ qemu-system-x86_64 -m 4G -bios OVMF.fd

■ OVMF.fd をダウンロードしてきた場合
$ qemu-system-x86_64 -m 4G -bios OVMF.fd へのパス指定
例）
$ qemu-system-x86_64 -m 4G -bios ~/OVMF.fd
```

"-m 4G"は QEMU 上の仮想マシンのメモリ設定です。4GB に設定している理由は本章最後のコラムで説明します。

■ 実機の場合

UEFI シェル内蔵の場合は PC 起動時に F2 キー等で表示される BIOS(UEFI) 設定画面から UEFI シェルを起動できるようです。

ここでは UEFI シェルを内蔵していない場合に、公開されている UEFI シェル実行バイナリを使用する方法を紹介します。UEFI シェル内蔵の場合も使える方法なので、「内蔵されているか分からない」・「内蔵されているが上手く起動できない」場合はこの方法を試してみてください。

UEFI シェルの実行バイナリ ("Shell_Full.efi") は以下からダウンロードできます。

- `https://github.com/tianocore/edk2/tree/master/EdkShellBinPkg/FullShell/X64`

ダウンロード後、UEFI ファームウェアが見つけられるように、起動用ストレージ (USB フラッシュメモリ等) へ配置します。"Shell_Full.efi"を起動用ストレージのリスト 1.1 の場所へ配置してください。

なお、UEFI ファームウェアは FAT あるいは GPT[*6]の第 1 パーティションを認識します。USB フラッシュメモリ等は購入時、FAT パーティションが一つのみの状態であるため特に問題は無いかと思います。UEFI 向けの GPT フォーマットの仕方は付録に記載しておりますので、必要であれば参照してください。

▼リスト 1.1　UEFI シェルバイナリの配置

[*6] "GUID Partition Table"と呼ばれるもので、UEFI 向けに新たに作られたものです。

```
起動用ストレージ/
└──── EFI/
    └── BOOT/
        └── BOOTX64.EFI  ← Shell_Full.efi をリネームして配置
```

そして、PC の BIOS(UEFI) 設定を変更し、作成した起動用ストレージからブートすると UEFI シェルが立ち上がり、以下のようにプロンプトが表示されます。

```
Shell>
```

memmap コマンドでメモリマップを確認する

"memmap"というコマンドでメモリマップを確認できます。

実行例 (QEMU)

```
Shell> memmap
Type      Start            End              #pages             Attributes
Available 0000000000000000-000000000009FFFF 00000000000000A0 000000000000000F
Available 0000000000100000-000000000081FFFF 0000000000000720 000000000000000F
BS_Data   0000000000820000-0000000000FFFFFF 00000000000007E0 000000000000000F
Available 0000000001000000-00000000BBFFFFFF 00000000000BB000 000000000000000F
BS_Data   00000000BC000000-00000000BC01FFFF 0000000000000020 000000000000000F
Available 00000000BC020000-00000000BE968FFF 0000000000002949 000000000000000F
BS_Data   00000000BE969000-00000000BE98DFFF 0000000000000025 000000000000000F
Available 00000000BE98E000-00000000BE9B2FFF 0000000000000025 000000000000000F
BS_Data   00000000BE9B3000-00000000BE9BFFFF 000000000000000D 000000000000000F
Available 00000000BE9C0000-00000000BE9C0FFF 0000000000000001 000000000000000F
BS_Data   00000000BE9C1000-00000000BEB17FFF 0000000000000157 000000000000000F
LoaderCode 00000000BEB18000-00000000BEC2EFFF 0000000000000117 000000000000000F
BS_Data   00000000BEC2F000-00000000BECE7FFF 00000000000000B9 000000000000000F
BS_Code   00000000BECE8000-00000000BEE3BFFF 0000000000000154 000000000000000F
RT_Data   00000000BEE3C000-00000000BEE4DFFF 0000000000000012 800000000000000F
BS_Data   00000000BEE4E000-00000000BFD4DFFF 0000000000000F00 000000000000000F
BS_Code   00000000BFD4E000-00000000BFECDFFF 0000000000000180 000000000000000F
RT_Code   00000000BFECE000-00000000BFEFDFFF 0000000000000030 800000000000000F
RT_Data   00000000BFEFE000-00000000BFF21FFF 0000000000000024 800000000000000F
Reserved  00000000BFF22000-00000000BFF25FFF 0000000000000004 000000000000000F
ACPIRec   00000000BFF26000-00000000BFF2DFFF 0000000000000008 000000000000000F
ACPI_NVS  00000000BFF2E000-00000000BFF31FFF 0000000000000004 000000000000000F
BS_Data   00000000BFF32000-00000000BFFCFFFF 000000000000009E 000000000000000F
RT_Data   00000000BFFD0000-00000000BFFFFFFF 0000000000000030 800000000000000F
Available 0000000100000000-000000013FFFFFFF 0000000000040000 000000000000000F
  Reserved  :           4 Pages (16,384)
  LoaderCode:         279 Pages (1,142,784)
  LoaderData:           0 Pages (0)
  BS_Code   :         724 Pages (2,965,504)
  BS_Data   :       6,624 Pages (27,131,904)
  RT_Code   :          48 Pages (196,608)
```

```
    RT_Data   :         102 Pages (417,792)
    ACPI Recl :           8 Pages (32,768)
    ACPI NVS  :           4 Pages (16,384)
    MMIO      :           0 Pages (0)
    Available :   1,040,687 Pages (4,262,653,952)
  Total Memory: 4095 MB (4,294,574,080 Bytes)
```

結果が 1 画面に収まらない場合、勝手にスクロールしてしまいますが、"PgUp"/"PgDn" キーで上下にスクロールできます。

"Available"と記載されている領域が使用可能なメモリ領域で、"#pages"列に各領域のサイズが記載されています。サイズはページ数なので、ページサイズ (4KB) を掛けた値がバイト数になります。

1.2.6 メモリマップの内容から poiboot.conf 作成

memmap コマンドで表示されるメモリマップの中から kernel.bin と fs.img それぞれの配置先として 2 つの Available 領域を選び、以下のように poiboot.conf へ記載します。

- 1 行目: kernel.bin のロード先アドレス (16 進数 16 文字)
- 2 行目: fs.img のロード先アドレス (16 進数 16 文字)

kernel.bin と fs.img それぞれの Available 領域としては以下を満たすものを選んでください。

- 1 行目: kernel.bin のロード先
  - 先頭アドレスから 1MB 先をスタックのベースアドレスにしているため、サイズが 1MB 以上であること
  - デフォルトの設定ファイルの値: 0000000000110000
- 2 行目: fs.img のロード先
  - 本書最後の方で 1 枚 2MB 弱の画像ファイルをいくつか配置するため、20MB 以上はあると良い
  - デフォルトの設定ファイルの値: 0000000100000000

以降、本書ではデフォルトの設定ファイルの値で poiboot が設定されているものとして説明します。

QEMU の"-m 4G"オプションの理由

-m が 4GB 以上の時、1GB 以上の Available な領域がアドレス"0x0000000100000000"に対応付けられるようになります。実機 (lenovo) でも同様であったので「fs.img のロード先」として 0x0000000100000000 が使えるよう QEMU には"-m 4G"のオプションを付けています。

ちなみに、ネット上で memmap の実機での実行結果をいくつか見てみると、"0x0000000100000000"に 1GB 以上の Available 領域が対応付けられるのはどのマシンでも共通な様です。

また、"0x0000000100000000"には「搭載メモリ - 3GB」のメモリが対応付けられているようです。QEMU や実機で memmap を試すと分かりますが、"0x0000000100000000"より上位 (小さいアドレス) に散り散りに対応付けられている Available 領域が 3GB 弱程度で、それらを引いた残りが"0x0000000100000000"に対応付けられているようです。(厳密には UEFI ファームウェアが使用する分も引いた残りですが、それらは 1GB もありません。)

そのため、-m オプションに指定するメモリ容量は 4GB 以上であれば QEMU のメモリ設定はいくつでも構わないです。本書で扱う範囲では fs.img のための領域としては 1GB もあれば十分です。そのため、本書では QEMU の-m オプションには 4GB を指定して、"0x0000000100000000"に 1GB 分の Available 領域が対応付けられるようにしています。

メモリマップ取得をブートローダーで行うには

本書では簡単のためにメモリマップを UEFI シェルから確認しましたが、ブートローダーから UEFI ファームウェアへ問い合わせてメモリマップ取得を行うこともできます。

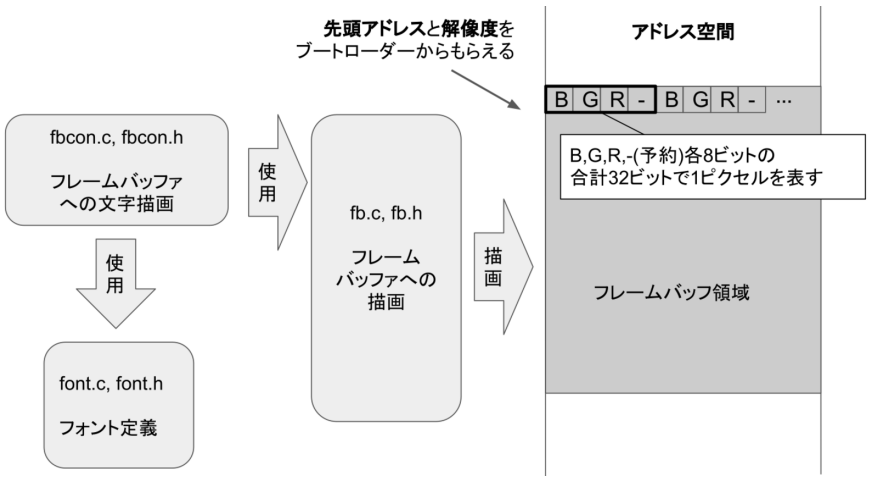
本書の範囲から外れるため説明しませんが、実装としてはメモリマップを画面にダンプする関数がサンプルソースの"A01_poiboot"ディレクトリ内の"libuefi/mem.c"に"dump_memmap"という関数名で定義していますので興味があれば見てみてください。

第 2 章

フレームバッファを制御する

この章ではフレームバッファへ書き込むプログラムの作成を通して本書での OS 自作の流れを紹介します。

この章では最終的に図 2.1 の構成でフレームバッファ関連の機能を実装します。



▲図 2.1　この章で作る機能の構成

図 2.1 の角丸の四角が作成するソースコードです。フレームバッファドライバ (fb.c と fb.h)、フレームバッファコンソール (fbcon.c と fbcon.h)、フォント (font.c と font.h) を作成します。

2.1 画面を単色で塗りつぶす

最初に行う"Hello world"的プログラムとして、画面を単色で塗りつぶしてみます。本書で使用するブートローダー"poiboot"はメモリへロードしたカーネルを実行する際、C 言語の引数としてフレームバッファの先頭アドレスを渡しますので、それを使用することで画面描画が行えます。

ここで紹介するプログラムはサンプルディレクトリ内の"020_fill"ディレクトリに格納しています。

2.2 ソースコードを書く

まずはソースコードを書きます。ソースコードは"main.c"の一つだけで、内容はリスト 2.1 の通りです。

▼**リスト** 2.1　020_fill/main.c

```
#define BG_RED          0
#define BG_GREEN        255
#define BG_BLUE                 0

struct pixelformat {
        unsigned char b;
        unsigned char g;
        unsigned char r;
        unsigned char _reserved;
};

struct framebuffer {
        struct pixelformat *base;
        unsigned long long size;
        unsigned int hr;
        unsigned int vr;
};

void start_kernel(void *_t __attribute__ ((unused)), struct framebuffer *fb,
                  void *_fs_start __attribute__ ((unused)))
{
        unsigned int x, y;
        struct pixelformat *p;

        for (y = 0; y < fb->vr; y++) {
                for (x = 0; x < fb->hr; x++) {
                        p = fb->base + (fb->hr * y) + x;
                        p->r = BG_RED;
                        p->g = BG_GREEN;
                        p->b = BG_BLUE;
                }
        }
```

```
        while (1);
}
```

start_kernel 関数がブートローダーからジャンプしてくる先の関数で、第 2 引数に"framebuffer"という構造体のポインタとしてフレームバッファの情報を渡してくれます。第 1 引数にもブートローダーから何か情報を渡しているのですが、本書では使わないので気にしなくて構いません[*1]。第 3 引数はブートローダーが配置したファイルシステムの先頭アドレスです。これについては後の章で使用します。

framebuffer 構造体の"base"メンバはフレームバッファの先頭アドレスです。ピクセルフォーマットは"pixelformat"構造体の通りで、「B(青)・G(緑)・R(赤)・Reserved(予約)」の順に各 1 バイトで並んでいます。加えて、framebuffer の"hr"メンバには画面の水平解像度、"vr"には垂直解像度が格納されていますので、start_kernel 関数の for の二重ループのコードで画面を単色で塗りつぶすことができます。ここでは緑一色に塗りつぶしています。

start_kernel 関数の戻り先は無いので、最後は無限ループで止めています。

2.3 リンカスクリプトを書く

ソースコードは作成できたので、poiboot がロードし実行できるフォーマットでカーネルバイナリ"kernel.bin"を生成するようリンカスクリプトを書きます。

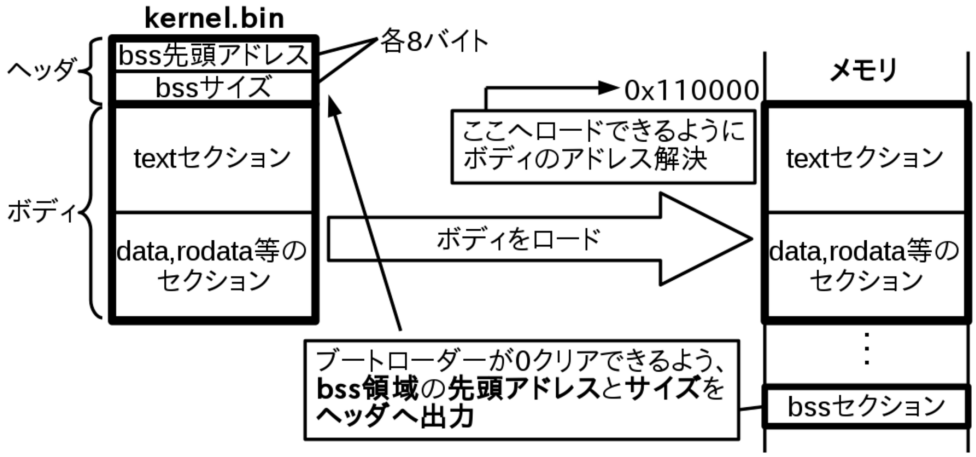
まず、kernel.bin はヘッダとボディで構成されます。ヘッダとボディそれぞれの内容を使用して poiboot が kernel.bin を実行するまでの流れは以下の通りです。

1. 設定ファイル"poiboot.conf"で指定された kernel.bin ロード先アドレスへボディをロード
2. ヘッダに記述された bss セクション[*2]先頭アドレスとサイズに従って bss セクションをゼロクリア
3. ロードしたボディの先頭へジャンプ

[*1] UEFI の SystemTable というものの先頭アドレスです。

[*2] 「セクション」とは、コンパイル過程のリンク時にバイナリデータを並べる単位です。リンカは「機械語コード」や「グローバル変数の初期値」といった領域 (セクション) 単位でバイナリを並べます。「機械語コード」を「text セクション」、「0 以外の初期値を持つグローバル変数の初期値」を「data セクション」等と呼び、「bss セクション」は「初期値無しあるいは 0 のグローバル変数」です。そのため、実行バイナリロード後にロード先の bss セクションはゼロクリアする必要があります。なお、厳密には「初期値無しグローバル変数」は COMMON セクションですが、bss とまとめてゼロクリアできるよう、Makefile で CFLAGS へ"-fno-common"を指定しています。

図示すると図 2.2 の通りです。



▲**図 2.2　kernel.bin のフォーマットと poiboot のロードの挙動**

poiboot が実行できるバイナリを生成するためにリンカスクリプトで考慮しなければならないことをまとめると以下の通りです。

- ヘッダへロード先の bss セクションの先頭アドレスとサイズを格納
- ボディの先頭へ text セクションを配置
- ボディの内容は、poiboot.conf で指定された kernel.bin ロード先アドレスに従ってアドレス解決する

ボディ先頭の text セクションについて、厳密には、加えて「text セクション先頭にカーネルのエントリ関数が配置されていること」も必要です。これについては main.c にエントリ関数のみ記載 (別の関数を記載するとしてもエントリ関数の後に関数定義する) とし、Makefile で ld へ指定するオブジェクトファイル順で main.o を 1 番目に指定することにします。

また、ボディの内容が poiboot.conf の設定に従ってアドレス解決する点については、ここでは簡単のため poiboot.conf に記載した kernel.bin ロード先アドレスと同じアドレスをリンカスクリプトにも記載することにします (本書では 0x00110000 を想定)。

以上の内容をリンカスクリプトとして記述するとリスト 2.2 の通りです。なお、リンカスクリプトは"kernel.ld"というファイル名で作成することとします。

▼**リスト 2.2　020_fill/kernel.ld**

```
OUTPUT_FORMAT("binary");

MEMORY
{
        exec_file(r) : ORIGIN = 0x00000000, LENGTH = 1m
        ram(wx)      : ORIGIN = 0x00110000, LENGTH = 960k
}

SECTIONS
{
        .header : {
                QUAD(__bss_start)
                QUAD(__bss_end - __bss_start)
        } > exec_file

        .body : {
                *(.text)

                *(.rodata)

                *(.data)

                __bss_start = .;
                *(.bss)
                __bss_end = .;

                *(.eh_frame)
        } > ram AT> exec_file
}
```

kernel.ld の内容を説明すると、まず OUTPUT_FORMAT() でいかなる実行ファイルフォーマットでもない単なるバイナリであることを示しています。

そして、"MEMORY{}"と"SECTIONS{}"では、上述したフォーマットとなるようにヘッダ (header) とボディ (body) の各種セクションの配置ルールを決めています。この時、header は実行ファイル (kernel.bin) にのみ配置され、メモリへはロードされないのに対し、body は実行ファイルに存在し、かつメモリへもロードされる、という違いがあります。

そのため、実行ファイルへの配置とアドレス解決の都合上、"MEMORY{}"内で"exec_file"(実行ファイル) と"ram"(メモリ) という 2 つの定義を用意しています。そして、"SECTIONS{}"内で、header は単純に exec_file に配置し (「> exec_file」の指定)、続く body はアドレス解決は ram に配置した想定で行いバイナリデータは exec_file に続いて配置を行っています (「> ram AT> exec_file」の指定)。

2.4 Makefile を書く

後は gcc コマンドでソースコードからオブジェクトファイルを生成し、ld コマンドで kernel.ld を使用してリンクを行うだけです。

gcc や ld コマンドは指定するオプションも多く、何度も手動でコマンドを実行するのは面倒なので、Makefile を作成します。内容はリスト 2.3 の通りです。

▼**リスト** 2.3　020_fill/Makefile

```
TARGET = kernel.bin
CFLAGS = -Wall -Wextra -nostdinc -nostdlib -fno-builtin -fno-common
LDFLAGS = -Map kernel.map -s -x -T kernel.ld

$(TARGET): main.o
        ld $(LDFLAGS) -o $@ $+

%.o: %.c
        gcc $(CFLAGS) -c -o $@ $<

clean:
        rm -f *~ *.o *.map $(TARGET)

.PHONY: clean
```

Makefile では特別なことはしていません。

デバッグ用にリンク時に"kernel.map"というファイル名でリンクマップ[*3]を生成しています。

2.5　コンパイルする

Makefile を作成しているため、コンパイルは"make"コマンドを実行するだけです。

ソースコード、リンカスクリプト、Makefile を配置したディレクトリへ移動し、以下のように"make"コマンドを実行してください。

```
$ make
```

make が成功すると"kernel.bin"が生成されています。

```
$ ls kernel.bin
kernel.bin
```

2.6　QEMU で実行する

QEMU には特定のディレクトリ以下をディスクと見なす機能があります。

[*3] ld によってマップされたシンボルの位置情報やセクションの位置情報などが書かれたテキストファイル

"fs"というディレクトリを作成し、1 章でダウンロード・展開したブートローダー (poiboot.efi) と kernel.bin をリスト 2.4 のように配置してください。

▼**リスト** 2.4　QEMU **実行時のファイル配置**

```
fs/
├── EFI/
│   └── BOOT/
│       └── BOOTX64.EFI   ← poiboot.efi をリネームして配置
└── kernel.bin
```

UEFI ファームウェアは起動ディスク第 1 パーティションに"EFI/BOOT/-BOOTX64.EFI"というファイル名でファイルが置いてあるとそれを見つけて実行してくれます。その後、poiboot が kernel.bin を見つける流れは先述の通りです。

ファイル配置後は以下のコマンドで実行できます。

```
$ qemu-system-x86_64 -bios OVMF.fd -hda fat:fs
```

"-hda"オプションに"fat:fs"のように指定すると、「fs ディレクトリ以下を FAT フォーマットされたディスク」とみなして実行してくれます。

また、"-bios"の"OVMF.fd"は、OVMF.fd を手動でダウンロードした場合は、"OVMF.fd"を配置したパスを指定してください。

QEMU を起動すると図 2.3 のように緑一色の画面が表示されます (白黒印刷なので印刷上は分かりませんが。。。)。

▲図 2.3　020_fill の QEMU 実行結果

終了する際は、QEMU のウィンドウごと X ボタンで終了してしまって構いません (特にストレージ等をマウントしている訳では無いので)。

kernel.bin 更新の都度、fs ディレクトリへファイルを配置して qemu コマンドを打つのも面倒なので、これらもリスト 2.5 の様に Makefile へ追記します。

▼リスト 2.5　QEMU 実行手順も Makefile へ追記

```
TARGET = kernel.bin
CFLAGS = -Wall -Wextra -nostdinc -nostdlib -fno-builtin -fno-common
LDFLAGS = -Map kernel.map -s -x -T kernel.ld

$(TARGET): main.o
        ld $(LDFLAGS) -o $@ $+

%.o: %.c
        gcc $(CFLAGS) -c -o $@ $<

# 追加（ここから）
run: $(TARGET)
        cp $(TARGET) ../fs/
        qemu-system-x86_64 -bios OVMF.fd -hda fat:../fs
# 追加（ここまで）

clean:
        rm -f *~ *.o *.map $(TARGET)

.PHONY: run clean  # run を追加
```

リスト 2.5 は、サンプルのディレクトリの一つ上の階層に fs ディレクトリがブートローダー配置済みの状態で存在していることを前提としています。

予め、020_fill の一つ上の階層にリスト 2.6 のようにディレクトリ構造を作っておいてください。

▼**リスト 2.6　Makefile の run ターゲット実行時のディレクトリ構造**

```
020_fill/
├── Makefile
├── kernel.ld
└── main.c
fs/
└── EFI/
    └── BOOT/
        └── BOOTX64.EFI
```

2.7　実機で試す

実機で試す際は、USB フラッシュメモリ等の PC の起動ディスクとして使用できるストレージへ fs ディレクトリ内のファイルをコピーし、コピーしたストレージから起動します。

なお、ストレージの第 1 パーティションは GPT あるいは FAT でフォーマットしておいてください。ただし、たいていの USB フラッシュメモリは購入時、FAT フォーマットされたパーティション 1 つの状態なので特に再フォーマットの必要は無いです。フォーマットする場合、GPT フォーマットの方法は参考までに本書末尾の付録に記載しています。

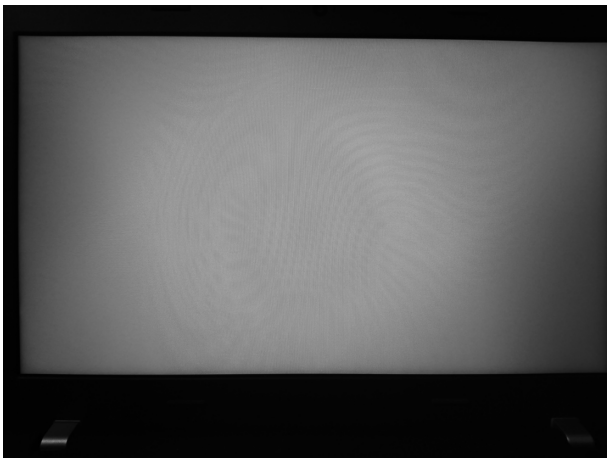
2.7.1　USB フラッシュメモリへファイル配置・起動

USB フラッシュメモリ第 1 パーティションへ fs ディレクトリ内のディレクトリ構造をまるごとコピーします。

コマンドとしては以下の通りです (USB フラッシュメモリ第 1 パーティションが"/dev/sdb1"として認識されているとします)。

```
$ sudo mount /dev/sdb1 /mnt
$ sudo cp -r fs/* /mnt/
$ sudo umount /mnt
```

その後、BIOS 設定で起動デバイスの優先順位設定を変更し、USB フラッシュメモリから起動してください。すると図 2.4 の様に動作確認できます (といってもこちらも白黒印刷なので印刷上は分かりませんが。。)。



▲図 2.4　020_fill の実機実行結果

シャットダウンなどの処理は無いので、電源ボタンで終了させてください (ディスク書き込み等はしていないので、何も害はありません)。

2.8　ソースコードを整理

最後に、ソースコードの見通しが悪くならないよう、ソースコードを整理します。

整理後のソースコードはサンプルディレクトリの"021_fill"ディレクトリに格納しています。

ここでは、fb.h と fb.c というソースファイルを作成し、フレームバッファに関する処理はそこへ移動します。fb.h の内容はリスト 2.7、fb.c の内容はリスト 2.8 の通りです。

▼リスト 2.7　021_fill/include/fb.h

```
#ifndef _FB_H_
#define _FB_H_

struct pixelformat {
        unsigned char b;
        unsigned char g;
        unsigned char r;
        unsigned char _reserved;
};

struct framebuffer {
        struct pixelformat *base;
        unsigned long long size;
        unsigned int hr;
```

```
        unsigned int vr;
};

void fb_init(struct framebuffer *_fb);
void set_bg(unsigned char r, unsigned char g, unsigned char b);
inline void draw_px(unsigned int x, unsigned int y,
                    unsigned char r, unsigned char g, unsigned char b);
inline void fill_rect(unsigned int x, unsigned int y,
                      unsigned int w, unsigned int h,
                      unsigned char r, unsigned char g, unsigned char b);
void clear_screen(void);

#endif
```

▼**リスト** 2.8　021_fill/fb.c

```
#include <fb.h>

struct framebuffer fb;
struct pixelformat color_bg;

void fb_init(struct framebuffer *_fb)
{
        fb.base = _fb->base;
        fb.size = _fb->size;
        fb.hr = _fb->hr;
        fb.vr = _fb->vr;
}

void set_bg(unsigned char r, unsigned char g, unsigned char b)
{
        color_bg.b = b;
        color_bg.g = g;
        color_bg.r = r;
}

inline void draw_px(unsigned int x, unsigned int y,
                    unsigned char r, unsigned char g, unsigned char b)
{
        struct pixelformat *p = fb.base;
        p += y * fb.hr + x;

        p->b = b;
        p->g = g;
        p->r = r;
}

inline void fill_rect(unsigned int x, unsigned int y,
                      unsigned int w, unsigned int h,
                      unsigned char r, unsigned char g, unsigned char b)
{
        unsigned int i, j;
        for (i = y; i < (y + h); i++)
                for (j = x; j < (x + w); j++)
                        draw_px(j, i, r, g, b);
}
```

```
void clear_screen(void)
{
        fill_rect(0, 0, fb.hr, fb.vr, color_bg.r, color_bg.g, color_bg.b);
}
```

一気にコード行数が増えた感じがありますが、先ほどのフレームバッファへの画面塗りつぶし処理を機能ごとに分けて関数化しているだけです。用意した関数は以下の 5 つです。

- fb_init
  - フレームバッファ情報を登録
- set_bg
  - 背景色を設定
- draw_px
  - 指定した座標 1 ピクセルを指定した色で描画する
- fill_rect
  - 座標・幅・高さで指定した領域を指定した色で塗りつぶす
- clear_screen
  - 背景色で画面を塗りつぶす

draw_px 関数や fill_rect 関数は他の関数からも何度も呼び出される関数なので、インライン関数化しています[*4]。

また、ヘッダファイルは include ディレクトリに配置し、"#include <>"で include できるようにコンパイルオプションへ追加します (後述)。

以上の fb.h・fb.c を使用すると、main.c はリスト 2.9 のように書けます。

▼リスト 2.9　021_fill/main.c

```
#include <fb.h>

void start_kernel(void *_t __attribute__ ((unused)), struct framebuffer *_fb,
                  void *_fs_start __attribute__ ((unused)))
{
        fb_init(_fb);
        set_bg(0, 255, 0);
        clear_screen();

        while (1);
```

[*4] GCC は頭が良いのか inline 指定しなくても draw_px と fill_rect 関数はインライン展開されていました。また、以降の章でも inline 指定の関数はいくつか登場しますが、どうも inline 指定していてもインライン展開されない場合もあるようです。

```
}
```

start_kernel 関数内が初期化処理っぽくなりました。また、start_kernel の第 2 引数は fb_init 関数へ渡すだけの一時変数になったため"_fb"という変数名へ変更しました。

最後に Makefile へ fb.o と include ディレクトリの指定を追加します (リスト 2.10)。

▼**リスト** 2.10　021_fill/Makefile

```
TARGET = kernel.bin
CFLAGS = -Wall -Wextra -nostdinc -nostdlib -fno-builtin -fno-common -Iinclude
#                                                                    ↑追加
LDFLAGS = -Map kernel.map -s -x -T kernel.ld

$(TARGET): main.o fb.o  # fb.oを追加
        ld $(LDFLAGS) -o $@ $+

%.o: %.c
        gcc $(CFLAGS) -c -o $@ $<

run: $(TARGET)
        cp $(TARGET) ../fs/
        qemu-system-x86_64 -bios OVMF.fd -hda fat:../fs

clean:
        rm -f *~ *.o *.map $(TARGET) include/*~  # "include/*~"を追加

.PHONY: run clean
```

これでコード整理は終了です。

ここで追加した fb.h と fb.c がフレームバッファのドライバに当たります。今後、このドライバを使用して、画面描画を行います。

また、以降はここで作成したソースコード構成を拡張していく形で機能拡張や機能追加を行っていきます。

2.9　画面に文字を表示する (フォント)

この章最後に、画面に文字を出してみます。

前著やそれ以前に QEMU 上で動く「BIOS+x86_32」を対象に書いた「Ohgami's Commentary on OS5」ではファームウェアや CRT コントローラのテキストモードを使用し、カーネル側でフォントを持つこと無く画面に文字を表示していました。

しかし、グラフィックモードをいずれ扱う際にはいずれフォントが必要になることと、64 ビットのアーキテクチャで CRT コントローラのテキストモード時の VRAM がアドレス空間のどこにマップされているかが分からなかったので、本著では早速フォントを用意してみます。

ソースコードはサンプルディレクトリの"022_font"に格納しています。

フォントの実装方法は色々あるかと思いますが、ここでは簡単に配列で実装してみます。今の GCC では連想配列の初期化のように配列を定義できます。フォントの定義は font.c というソースコードへ行うことにします。font.c の内容はリスト 2.11 の通りです。

▼**リスト** 2.11　022_font/font.c(**一部**)

```
#include <font.h>

const unsigned char font_bitmap[][FONT_HEIGHT][FONT_WIDTH] = {
        [' '] = {
                {0,0,0,0,0,0,0,0},
                {0,0,0,0,0,0,0,0},
                {0,0,0,0,0,0,0,0},
                {0,0,0,0,0,0,0,0},
                {0,0,0,0,0,0,0,0},
                {0,0,0,0,0,0,0,0},
                {0,0,0,0,0,0,0,0},
                {0,0,0,0,0,0,0,0},
                {0,0,0,0,0,0,0,0},
                {0,0,0,0,0,0,0,0}
        },
        ['!'] = {
                {0,0,0,0,0,0,0,0},
                {0,0,0,0,1,0,0,0},
                {0,0,0,0,1,0,0,0},
                {0,0,0,0,1,0,0,0},
                {0,0,0,0,1,0,0,0},
                {0,0,0,0,1,0,0,0},
                {0,0,0,0,1,0,0,0},
                {0,0,0,0,0,0,0,0},
                {0,0,0,0,1,0,0,0},
                {0,0,0,0,0,0,0,0}
        },
        /* ・・・ 省略 ・・・ */
        ['0'] = {
                {0,0,0,0,0,0,0,0},
                {0,0,1,1,1,1,0,0},
                {0,1,0,0,0,1,1,0},
                {0,1,0,0,1,0,1,0},
                {0,1,0,0,1,0,1,0},
                {0,1,0,1,0,0,1,0},
                {0,1,0,1,0,0,1,0},
                {0,1,1,0,0,0,1,0},
                {0,0,1,1,1,1,0,0},
                {0,0,0,0,0,0,0,0}
        },
        ['1'] = {
                {0,0,0,0,0,0,0,0},
                {0,0,0,0,1,0,0,0},
                {0,0,0,1,1,0,0,0},
                {0,0,0,0,1,0,0,0},
                {0,0,0,0,1,0,0,0},
                {0,0,0,0,1,0,0,0},
                {0,0,0,0,1,0,0,0},
                {0,0,0,0,1,0,0,0},
```

```
                {0,0,1,1,1,1,0,0},
                {0,0,0,0,0,0,0,0}
        },
        /* ・・・ 省略 ・・・ */
        ['A'] = {
                {0,0,0,0,0,0,0,0},
                {0,0,0,1,1,0,0,0},
                {0,0,0,1,1,0,0,0},
                {0,0,1,0,0,1,0,0},
                {0,0,1,0,0,1,0,0},
                {0,1,0,0,0,0,1,0},
                {0,1,1,1,1,1,1,0},
                {0,1,0,0,0,0,1,0},
                {0,1,0,0,0,0,1,0},
                {0,0,0,0,0,0,0,0}
        },
        ['B'] = {
                {0,0,0,0,0,0,0,0},
                {0,1,1,1,1,1,0,0},
                {0,1,0,0,0,0,1,0},
                {0,1,0,0,0,0,1,0},
                {0,1,1,1,1,1,0,0},
                {0,1,0,0,0,0,1,0},
                {0,1,0,0,0,0,1,0},
                {0,1,0,0,0,0,1,0},
                {0,1,1,1,1,1,0,0},
                {0,0,0,0,0,0,0,0}
        },
        /* ・・・ 省略 ・・・ */
};
```

リスト 2.11 で定義している font_bitmap 配列の宣言や、1 文字の幅 (FONT_WIDTH) と高さ (FONT_HEIGHT) は include/font.h に書いておきます。内容はリスト 2.12 の通りです。

▼**リスト** 2.12　022_font/include/font.h

```
#ifndef _FONT_H_
#define _FONT_H_

#define FONT_WIDTH     8
#define FONT_HEIGHT    10

extern const unsigned char font_bitmap[][FONT_HEIGHT][FONT_WIDTH];

#endif
```

このようにフォントのビットマップを定義しておくことで、先ほど実装した draw_px 関数を使用して画面に文字を出すことができるようになります。

現状の draw_px 関数は引数に色を指定していますが、1 つの文字を描画している最中に色を変えることは考えないので、背景色同様に、描画色も予め設定しておくことにし

ます。

フレームバッファドライバ (fb.c) へ描画色 (color_fg) と設定関数 (set_fg 関数)、描画色で 1 ピクセルを描画する関数 (draw_px_fg) の定義を追加します。追加後の fb.c はリスト 2.13 の通りです。

▼**リスト** 2.13　022_font/fb.c

```
#include <fb.h>

struct framebuffer fb;
struct pixelformat color_fg;    /* 追加 */
struct pixelformat color_bg;

/* ・・・ 省略 ・・・ */

/* 追加（ここから） */
void set_fg(unsigned char r, unsigned char g, unsigned char b)
{
        color_fg.b = b;
        color_fg.g = g;
        color_fg.r = r;
}
/* 追加（ここまで） */

void set_bg(unsigned char r, unsigned char g, unsigned char b)
{
/* ・・・ 省略 ・・・ */

/* 追加（ここから） */
inline void draw_px_fg(unsigned int x, unsigned int y)
{
        draw_px(x, y, color_fg.r, color_fg.g, color_fg.b);
}
/* 追加（ここまで） */
```

また、include/fb.h へ set_fg 関数のプロトタイプ宣言追加と、fb の extern 宣言追加[*5]も行います (リスト 2.14)。

▼**リスト** 2.14　022_font/include/fb.h

```
/* ・・・ 省略 ・・・ */

extern struct framebuffer fb;   /* 追加 */

void fb_init(struct framebuffer *_fb);
void set_fg(unsigned char r, unsigned char g, unsigned char b); /* 追加 */
void set_bg(unsigned char r, unsigned char g, unsigned char b);
inline void draw_px(unsigned int x, unsigned int y,
                    unsigned char r, unsigned char g, unsigned char b);
```

[*5] 改行処理のためにフレームバッファの垂直解像度 (struct fb の vr メンバ) を参照するため

```
inline void draw_px_fg(unsigned int x, unsigned int y);         /* 追加 */
/* ・・・ 省略 ・・・ */
```

font_bitmap 配列と draw_px_fg 関数を使用すると、指定された一文字を描画する putc 関数はリスト 2.15 の様に定義できます。

▼**リスト** 2.15　putc **関数**

```
unsigned int cursor_x = 0, cursor_y = 0;

void putc(char c)
{
        unsigned int x, y;

        /* カーソル座標 (cursor_x,cursor_y) へ文字を描画 */
        for (y = 0; y < FONT_HEIGHT; y++)
                for (x = 0; x < FONT_WIDTH; x++)
                        if (font_bitmap[(unsigned int)c][y][x])
                                draw_px_fg(cursor_x + x, cursor_y + y);

        /* カーソル座標の更新 */
        cursor_x += FONT_WIDTH;
        if ((cursor_x + FONT_WIDTH) >= fb.hr) {
                cursor_x = 0;
                cursor_y += FONT_HEIGHT;
                if ((cursor_y + FONT_HEIGHT) >= fb.vr) {
                        cursor_x = cursor_y = 0;
                        clear_screen();
                }
        }
}
```

リスト 2.15 ではカーソル座標をグローバル変数として用意し、putc 関数ではカーソル座標へ文字を描画後、カーソル座標の更新を行っています。フォントのビットマップを ASCII 文字の値を添え字とした配列として用意しているため、引数で指定された文字 (char c) をそのままフォント配列の添字に使用しています。

あと改行文字の処理を加え、putc 関数は完成とします。フレームバッファコンソールとして fbcon.c というソースコードを追加し、putc 関数はこちらへ追加することにします。fbcon.c の内容はリスト 2.16 の通りです。

▼**リスト** 2.16　022_font/fbcon.c

```
#include <fbcon.h>
#include <fb.h>
#include <font.h>

unsigned int cursor_x = 0, cursor_y = 0;
```

```
void putc(char c)
{
        unsigned int x, y;

        switch(c) {
        case '\r':
                cursor_x = 0;
                break;

        case '\n':
                cursor_y += FONT_HEIGHT;
                if ((cursor_y + FONT_HEIGHT) >= fb.vr) {
                        cursor_x = cursor_y = 0;
                        clear_screen();
                }
                break;

        default:
                /* カーソル座標 (cursor_x,cursor_y) へ文字を描画 */
                for (y = 0; y < FONT_HEIGHT; y++)
                        for (x = 0; x < FONT_WIDTH; x++)
                                if (font_bitmap[(unsigned int)c][y][x])
                                        draw_px_fg(cursor_x + x, cursor_y + y);

                /* カーソル座標の更新 */
                cursor_x += FONT_WIDTH;
                if ((cursor_x + FONT_WIDTH) >= fb.hr) {
                        cursor_x = 0;
                        cursor_y += FONT_HEIGHT;
                        if ((cursor_y + FONT_HEIGHT) >= fb.vr) {
                                cursor_x = cursor_y = 0;
                                clear_screen();
                        }
                }
        }
}
```

ついでに文字列を出力する puts 関数も fbcon.c へ追加しておきます (リスト 2.17)。

▼**リスト** 2.17　022_font/fbcon.c(puts **関数**)

```
/* ・・・ 省略 ・・・ */

void putc(char c)
{
        /* ・・・ 省略 ・・・ */
}

/* 追加（ここから） */
void puts(char *s)
{
        while (*s != '\0')
                putc(*s++);
}
/* 追加（ここまで） */
```

putc 関数と puts 関数を呼び出せるよう、include/fbcon.h をリスト 2.18 の様に作成します。

▼**リスト** 2.18　022_font/include/fbcon.h

```
#ifndef _FBCON_H_
#define _FBCON_H_

void putc(char c);
void puts(char *s);

#endif
```

それでは、set_fg 関数と puts 関数を使用して"HELLO WORLD!"を出力するよう main.c を書き替えてみます (リスト 2.19)。

▼**リスト** 2.19　022_font/main.c

```
#include <fb.h>
#include <fbcon.h>

void start_kernel(void *_t __attribute__ ((unused)), struct framebuffer *fb,
                  void *_fs_start __attribute__ ((unused)))
{
        fb_init(fb);
        set_fg(255, 255, 255);  /* 追加 */
        set_bg(0, 70, 250);     /* 変更 */
        clear_screen();

        puts("HELLO WORLD!");   /* 追加*/

        while (1);
}
```

最後に、Makefile へ font.c と fbcon.c をコンパイルの対象として追加します (リスト 2.20)。

▼**リスト** 2.20　022_font/Makefile

```
TARGET = kernel.bin
CFLAGS = -Wall -Wextra -nostdinc -nostdlib -fno-builtin -fno-common -Iinclude
LDFLAGS = -Map kernel.map -s -x -T kernel.ld

$(TARGET): main.o fbcon.o fb.o font.o       # fbcon.o と font.o を追加
        ld $(LDFLAGS) -o $@ $+

# ・・・ 省略 ・・・
```

■メモ: GNU ld へ渡すオブジェクトファイル順について

ld へ渡すオブジェクトファイルの順序として GNU ld の場合、「オブジェクトファイルのシンボル名解決を最初の方 (左側) から順に行い、解決できなかったシンボルを後続のオブジェクトファイルから探す」という挙動となります。そのため、色々なオブジェクトファイルを参照する main.o を一番最初のオブジェクトファイルとして渡し、参照されるだけである font.o は一番最後に ld へ渡しています。

実行すると図 2.5 の様に"HELLO WORLD!"と表示されます。

▲図 2.5　022_font の実行結果

OS のシステムコンソールっぽく (?)、青背景に白文字にしてみました[*6]。色はお好きなように変えてみてください。

また、本書で用意している font.c にはアルファベットは大文字しか用意していないので、Hello world も全て大文字で出しています。小文字の追加や、あるいはフォントサイズ等も変更して独自のフォントを実装してみたりすると自作 OS に愛着がでてきて良いです。

ここまでで、フレームバッファ出力に関する実装は完了です。

[*6] 白黒印刷なのでこちらも印刷上は分かりませんが。。

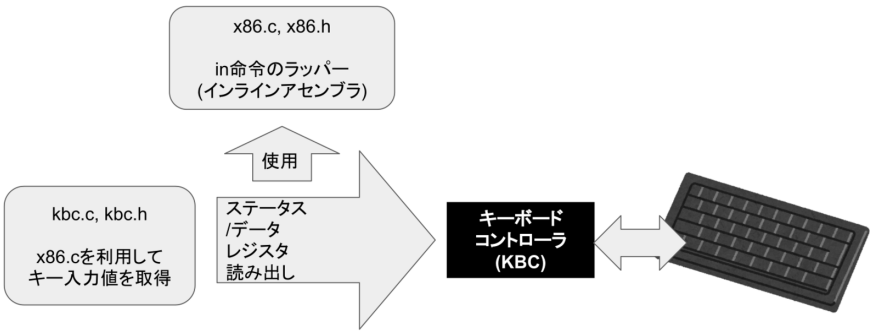
第 3 章

キーボード入力

文字を出力することはできたので、次はキーボードの文字入力を取得してみます。本書ではキーボードコントローラ (KBC) を使用してキー入力の取得を行います。

この章を通して作成するソースコードはサンプルディレクトリの"030_keyinput_polling"に格納しています。また、文中では前章最後の"022_font"のサンプルへ機能拡張・機能追加を行っていく形で説明していきます。

加えて、この章で実装する KBC 関連の機能の構成は図 3.1 の通りです。



▲図 3.1　この章で作る機能の構成

kbc.c と kbc.h が KBC のデバイスドライバに当たるものです。また、x86.c と x86.h は x86 CPU 依存の処理をまとめたアーキテクチャ依存部です。x86.c と x86.h へはこの章以降も適宜関数を追加していきます。

3.1 キーボードコントローラ (KBC) の使い方

キーボードコントローラは 32 ビット (そしてそれ以前) のアーキテクチャから存在するコントロール IC で、その名の通りキーボードの制御を行ってくれます。昔は Intel の 8042(i8042) という IC が KBC としてマザーボードに搭載されていましたが、集積度が上がり、現在は互換の機能がひとつの SoC[*1]内に入っています。

どんな制御 IC でもそうですが、KBC もレジスタ経由で操作します。各レジスタにはアドレスが割り当てられており、特定のアドレスへ読み書きすることで制御 IC のレジスタへアクセスできます。ただし、Intel の場合、周辺の制御 IC のレジスタは通常のメモリとは別のアドレス空間であり、io 命令 (in 命令と out 命令) を使用してアクセスします。

3.2 ステータスレジスタとデータレジスタ

KBC には現在の状態を取得するためのステータスレジスタと、入力されたキーのデータ (キーコード) を取得するためのデータレジスタがあります。

KBC のステータスレジスタとデータレジスタの IO アドレスと読み出せる値の意味は表 3.1 の通りです。

▼表 3.1 KBC のレジスタの IO アドレスと読み出せる値の意味

IO アドレス	レジスタ名と読み出した値の意味
0x0060	データレジスタ
	bit7：押下状態 (make=0 / brake=1)
	bit6-0: キーコード
0x0064	ステータスレジスタ
	bit7-4: 予約
	bit3：KBC コマンド/パラメータ書込フラグ
	bit2：予約
	bit1：IBF(入力バッファフル)
	bit0：OBF(出力バッファフル)

表 3.1 の 2 つのレジスタのビットフィールドの中で、ステータスレジスタ (0x0064) の bit0 と、データレジスタ (0x0060) の全ビットを使用します。

ステータスレジスタの OBF と IBF は、少しわかりにくいのですが、KBC 自身から見た入力/出力であるため、KBC という IC に対して外部に存在する x86 CPU は KBC

[*1] System on Chip の略。一つのチップの中に複数の IC に相当する機能が集積されている。

の出力バッファからデータを読み出す事になります。そのため、KBC の OBF(出力バッファフル) を確認してデータ読み出しを行います。

データレジスタの押下状態 (bit7) は、指でキーを押している状態を"make"、キーから指を離した状態を"brake"と呼んでいます。そのため、キーを押しっぱなしにしている間はデータレジスタの bit7 は 0(make) で読み出せ、キーから指を離すと一度だけ bit7 が 1(brake) のデータがデータレジスタから読み出せます。

3.3 インラインアセンブラ

in 命令はアセンブラで記述しなければなりませんが、本書では極力 C 言語で記述するためにインラインアセンブラを使用してみます。

まず、in 命令を使用した IO アドレスのレジスタ読み出しはアセンブラでリスト 3.1 のように記述します。

▼リスト 3.1　in 命令を使用したステータスレジスタ読み出し例 (アセンブラ)

```
        movw    $0x0064, %dx
        inb     %dx, %al
```

リスト 3.1 について、アセンブラでは movb や inb といった命令に当たる部分を「オペコード」、続く"$0x0064, %dx"等を「オペランド」と呼びます。

オペコードには mov や in といった命令に'w' や'b' といった一文字が付いていますが、これはデータサイズを示しています。'w' は 2 バイト、'b' は 1 バイトで、他にも'q'(8 バイト)、'l'(4 バイト) があります。

リスト 3.1 では、ステータスレジスタのアドレス (0x0064) を mov 命令で dx レジスタへ格納し、in 命令で dx レジスタが指す IO アドレスの値 (KBC ステータスレジスタの値) を al レジスタへ格納します。

リスト 3.1 をインラインアセンブラで記述するとリスト 3.2 の様になります。

▼リスト 3.2　インラインアセンブラでのステータスレジスタ読み出し例

```
unsigned short ioaddr = 0x0064;
unsigned char value;
asm volatile ("inb %[ioaddr], %[value]"
              : [value]"=a"(value) : [ioaddr]"d"(ioaddr));
```

リスト 3.2 について、"asm"がインラインアセンブラであることを示す指定です。"asm"というシンボルを別の用途で使用したい場合等は"___asm___"と書くこともできます (そのように書いているコードも多いです)。そして、コンパイラの最適化の影響を受けない

様、"volatile"を付けています。

リスト 3.2 の"inb %[ioaddr], %[value]"が元のアセンブラ (リスト 3.1) の"inb %dx, %al"に当たる箇所です。なんとなく分かるかと思いますが、変数"ioaddr"の IO アドレスにあるレジスタの値を変数"value"へ読み出しています。

ちゃんと説明すると、インラインアセンブラ内の要素はコロン (:) 区切りでリスト 3.3 のように並んでいます[*2]。

▼リスト 3.3　インラインアセンブラの構成要素

```
asm volatile（アセンブラテンプレート : 出力オペランド : 入力オペランド）
```

そして、出力オペランドと入力オペランドのフォーマットはリスト 3.4 の通りです (入力オペランドと出力オペランドの違いは'=' が付くかどうかだけです)。

▼リスト 3.4　入力/出力オペランドのフォーマット

```
■ 出力オペランド
=[アセンブラから参照する名前]"使用するレジスタ等の指定"（変数名）
■ 入力オペランド
[アセンブラから参照する名前]"使用するレジスタ等の指定"（変数名）
```

リスト 3.4 の"使用するレジスタ等の指定"には、in 命令の仕様の都合上、入力オペランドには"d"のレジスタを、出力オペランドには"a"のレジスタを指定しています。

[*2] 入力オペランドの後ろにもう一つコロンで区切る事で「使用するレジスタのリスト」を指定することもできますが、本書では使用しないため特に説明しません。

AX、EAX、RAX 等のレジスタ名について

x86 の汎用レジスタには、32 ビットでは EAX や EBX、64 ビットでは RAX や RBX のように頭に'E'、あるいは'R' が付き、最後に'X' で終わる名前のレジスタがいくつかあります。

この様な命名には歴史的な経緯があります。そもそも、初めは A や B といった 8 ビットのレジスタがあり、CPU の 16 ビットへの進化に併せて 8 ビットから 16 ビット拡張され"extended"の意味で'X' を付け、16 ビットレジスタは AX レジスタ、BX レジスタといった名前となりました。その後、32 ビットへのさらなる拡張に伴い、同様に"extended"の意味で'E' が頭に付けられ 32 ビットレジスタは EAX、EBX といった名前となりましたが、64 ビットでは流石に"extended"から文字を取るのに限界を感じたのか RAX、RBX という名前になっています[a][b]。

同じアルファベットに対して'E' や'R'、'X' が付いたレジスタはアクセスするサイズが違うだけで同じレジスタを指していて、RAX の下位 32 ビットを EAX でアクセスでき、EAX の下位 16 ビットを AX でアクセスできます。AX の上位と下位 8 ビットはそれぞれ"AH"・"AL"という名前でアクセスできます ('A' というレジスタ名は無くなりました)[c]。

[a] 書籍"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！ "より

[b] 64 ビットモードでは汎用レジスタ自体も 16 個へ増えており、増えた分は"R8"～"R15"という名前になっています。そのため、RAX 等の'R' は単に"Register"を指すものと思われます。

[c] R8～R15 の様な 64 ビットレジスタは R8D～R15D という名前で下位 32 ビットアクセスできます。(16 ビット単位/8 ビット単位のレジスタ名はありません)

3.4 キー入力を取得する

それでは、キーが入力されたかどうかをソフトウェア的にステータスレジスタを監視する方法 (ポーリング) で実装してみます。(ハードウェアによる割り込みを使用する方法は次章で紹介します。)

まずは、「指定した IO アドレスのレジスタから値を取得する」処理を C 言語から汎用的に呼べるように関数化します。本書では、インラインアセンブラを使用して"io_read"という関数を用意することにします。x86 CPU に依存した関数群は"x86.c"というソースファイルにまとめることにし、対応して"x86.h"というヘッダファイルも用意します (リス

ト 3.5、リスト 3.6)。

▼リスト 3.5　030_keyinput_polling/include/x86.h

```
#ifndef _X86_H_
#define _X86_H_

inline unsigned char io_read(unsigned short addr);

#endif
```

▼リスト 3.6　030_keyinput_polling/x86.c

```
#include <x86.h>

inline unsigned char io_read(unsigned short addr)
{
        unsigned char value;
        asm volatile ("inb %[addr], %[value]"
                      : [value]"=a"(value) : [addr]"d"(addr));
        return value;
}
```

それでは、io_read 関数を使用して、データレジスタの取得を行う"get_kbc_data"関数と、get_kbc_data 関数を使用してキーコードの取得を行う"get_keycode"関数を作成します。KBC に依存した処理のため"kbc.c"で定義することにします。内容はリスト 3.7 の通りです。

▼リスト 3.7　030_keyinput_polling/kbc.c(get_kbc_data() と get_keycode())

```
#define KBC_DATA_ADDR          0x0060
#define KBC_DATA_BIT_IS_BRAKE  0x80
#define KBC_STATUS_ADDR                0x0064
#define KBC_STATUS_BIT_OBF     0x01

unsigned char get_kbc_data(void)
{
        /* ステータスレジスタの OBF がセットされるまで待つ */
        while (!(io_read(KBC_STATUS_ADDR) & KBC_STATUS_BIT_OBF));

        return io_read(KBC_DATA_ADDR);
}

unsigned char get_keycode(void)
{
        unsigned char keycode;

        /* make 状態（brake ビットがセットされていない状態）まで待つ */
        while ((keycode = get_kbc_data()) & KBC_DATA_BIT_IS_BRAKE);
```

```
        return keycode;
}
```

ポーリング (ソフトウェア的な待機処理) を行っているのが正にリスト 3.7 の get_kbc_data 関数と get_keycode 関数それぞれの while です。

また、リスト 3.7 の get_keycode 関数では make 状態のキーコードを待ちます。brake 状態のキーコードを待つようにしたい場合は while 内の条件を反転させてください。

キーコードを取得できたので、キーコードを ASCII へ変換し、画面へ表示してみます。キーコードを ASCII へ変換する配列"keymap"と、キー入力 1 文字を取得する"getc"関数、そして各種ヘッダの include を"kbc.c"へ追加します (リスト 3.8)。

▼**リスト** 3.8　030_keyinput_polling/kbc.c

```
#include <kbc.h>        /* 追加 */
#include <x86.h>        /* 追加 */

#define KBC_DATA_ADDR          0x0060
#define KBC_DATA_BIT_IS_BRAKE  0x80
#define KBC_STATUS_ADDR                0x0064
#define KBC_STATUS_BIT_OBF     0x01

/* 追加（ここから） */
const char keymap[] = {
        0x00, ASCII_ESC, '1', '2', '3', '4', '5', '6',
        '7', '8', '9', '0', '-', '^', ASCII_BS, ASCII_HT,
        'q', 'w', 'e', 'r', 't', 'y', 'u', 'i',
        'o', 'p', '@', '[', '\n', 0x00, 'a', 's',
        'd', 'f', 'g', 'h', 'j', 'k', 'l', ';',
        ':', 0x00, 0x00, ']', 'z', 'x', 'c', 'v',
        'b', 'n', 'm', ',', '.', '/', 0x00, '*',
        0x00, ' ', 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, '7',
        '8', '9', '-', '4', '5', '6', '+', '1',
        '2', '3', '0', '.', 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, '_', 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, '\\', 0x00, 0x00
};
/* 追加（ここまで） */

/* ・・・ 省略 ・・・ */

/* 追加（ここから） */
char getc(void)
{
        return keymap[get_keycode()];
}
/* 追加（ここまで） */
```

そして、kbc.c の getc 関数と、keymap 配列で使用している ASCII_ESC 等の ASCII 制御文字は外部からも参照できるよう"kbc.h"に宣言と定義を用意します (リスト 3.9)。

▼**リスト** 3.9　030_keyinput_polling/include/kbc.h

```
#ifndef _KBC_H_
#define _KBC_H_

#define ASCII_ESC      0x1b
#define ASCII_BS       0x08
#define ASCII_HT       0x09

char getc(void);

#endif
```

ここまでで、キー入力取得に必要な道具は揃いました。ここでは main.c を改造して、キー入力取得のサンプルとしてはよくある「エコーバック[*3]」を作ってみます。改造後の main.c はリスト 3.10 の通りです。

▼**リスト** 3.10　030_keyinput_polling/main.c

```
#include <fb.h>
#include <fbcon.h>
#include <kbc.h>        /* 追加 */

void start_kernel(void *_t __attribute__ ((unused)), struct framebuffer *fb,
                  void *_fs_start __attribute__ ((unused)))
{
        fb_init(fb);
        set_fg(255, 255, 255);
        set_bg(0, 70, 250);
        clear_screen();

        /* 変更箇所（ここから） */
        while (1) {
                char c = getc();
                if (('a' <= c) && (c <= 'z'))
                        c = c - 'a' + 'A';
                else if (c == '\n')
                        putc('\r');
                putc(c);
        }
        /* 変更箇所（ここまで） */
}
```

現状のフォントの都合上、アルファベットは大文字しか表示できないので、リスト 3.10

[*3] 入力された文字をそのまま画面へ表示するプログラム

では入力された文字がアルファベットの場合、大文字に変換しています。また、Enter を押下すると'\n'(LF) を受け取るので、'\n' 取得時は追加で'\r'(CR) も出力しています。

最後に今回追加したソースファイルを Makefile へ反映します (リスト 3.11)。

▼**リスト** 3.11　030_keyinput_polling/Makefile

```
TARGET = kernel.bin
CFLAGS = -Wall -Wextra -nostdinc -nostdlib -fno-builtin -fno-common -Iinclude
LDFLAGS = -Map kernel.map -s -x -T kernel.ld

$(TARGET): main.o fbcon.o fb.o font.o kbc.o x86.o  # kbc.o と x86.o を追加
        ld $(LDFLAGS) -o $@ $+

# ・・・ 省略 ・・・
```

ここまででソースコードの追加・変更は終わりです。実行すると、図 3.2 の様に入力した文字が画面に表示されます。

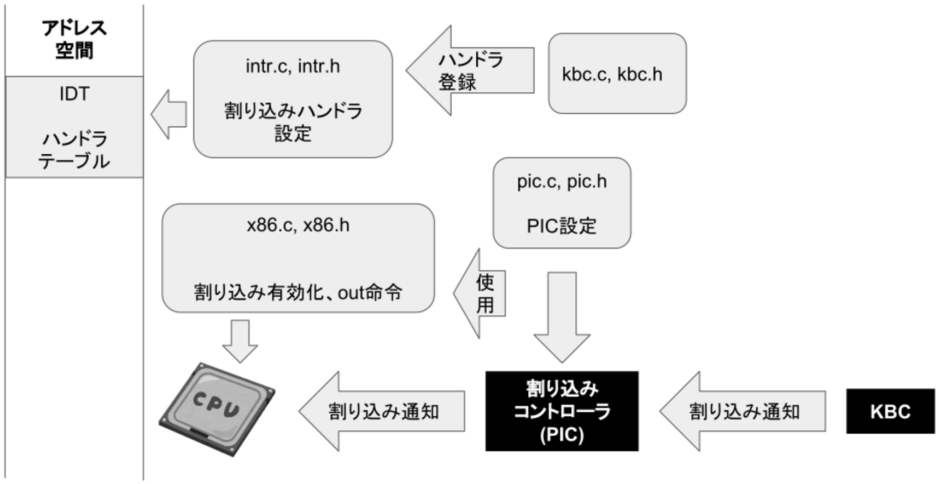
▲**図** 3.2　030_keyinput_polling **の実行結果**

第 4 章

割り込みを使う

次は割り込みでキーボード入力を取得してみます。割り込みも割り込みコントローラ (PIC、Programmable Interrupt Controller) という外部 IC[*1]の機能を利用します。ただし、割り込みの場合、CPU 側でも設定が必要です。

この章では図 4.1 の構成で割り込み関連の機能の実装を行います。



▲図 4.1　この章で作る機能の構成

この章では CPU の割り込み設定方法と PIC の使い方を説明し、前章のキー入力を割り

[*1] といっても今や KBC と同じく CPU に集積されています。

込みを使用して行うようにしてみます。章全体を通してサンプルディレクトリ"040_intr"の内容を説明します。

4.1 CPU: セグメンテーション設定

セグメンテーションとは、メモリ空間を分割して管理する方法の一つで、「セグメント」と呼ぶ単位で管理します。各セグメントは「開始アドレス (Base Address)」と「セグメントサイズ (Segment Limit)」を持ち、セグメント毎に「コード (機械語) 用である」とか「データ用である」とか、「カーネルモードでしかアクセスを許さない」といった「属性」を設定できます。

x86 CPU には、命令実行時のコード・データ・スタックについて、それぞれどのセグメントを参照するのかを指定するレジスタとして"CS(Code Segment)"・"DS(Data Segment)"・"SS(Stack Segment)"があるのですが、「IA-32e モード (64 ビットのモード) では一般的には無効[*2]」とのことです。

ただ、「完全には無効ではない[*3]」旨の括弧書きもあるため、一応、初期化を行います。ただし、メモリ管理方法は今や「ページング」と呼ばれる方法が主流であり、セグメントについて詳しい説明は行いません (あまり実りは無いので)。

セグメンテーションの初期化に当たって行うことは以下の通りです。

1. GDT の定義
2. レジスタ GDTR の設定 (lgdt 命令)
3. レジスタ CS・DS・SS の設定

4.1.1 GDT の定義

GDT(Global Descriptor Table) とは、セグメントを定義する"セグメントデスクリプタ (セグメント記述子)"の表です。メモリ上に用意しておき、GDTR というレジスタへ用意した GDT の先頭アドレスを設定する事で CPU は GDT の場所を知ります。

本書では GDT を実行時に変化させる事は無いので、リスト 4.1 の様に const の配列で定義してしまいます。

▼リスト 4.1　GDT の定義

[*2] 3.2.4 Segmentation in IA-32e Mode - Intel 64 and IA-32 Architectures Software Developer's Manual

[*3] 同じく、3.2.4 Segmentation in IA-32e Mode - Intel 64 and IA-32 Architectures Software Developer's Manual

```
const unsigned long long gdt[] = {
        0x0000000000000000,     /* NULL descriptor */
        0x00af9a000000ffff,     /* base=0, limit=4GB, mode=code(r-x),kernel */
        0x00cf93000000ffff      /* base=0, limit=4GB, mode=data(rw-),kernel */
};
```

リスト 4.1 では 3 つのセグメントデスクリプタを定義しています。GDT の一番最初のデスクリプタは 0(NULL) で定義することが決められているのでその様にしています。2 つ目は Code Segment を、開始アドレス 0、セグメントサイズ 4GB のカーネルモード用として定義しています。3 つ目は Data Segment で、開始アドレスとセグメントサイズとモードは Code Segment と同じです[*4]。セグメントサイズの 4GB は設定できる最大値です[*5]。

4.1.2 レジスタ GDTR の設定 (lgdt 命令)

定義した GDT の先頭アドレスを"GDTR(GDT Register)"へ設定することで CPU へ GDT の情報を伝えます。そのために使用する命令が"lgdt(Load GDT)"です。

GDTR には GDT の先頭アドレス (Base Address) とサイズ (Limit) を設定します。GDTR へ格納できるフォーマットでデータ列を用意しておき、作成したデータ列の先頭アドレスを指定して lgdt 命令を実行すると、データ列を GDTR へ設定してくれます (リスト 4.2)。

▼**リスト** 4.2　GDTR **の設定例**

```
unsigned long long gdtr[2];
gdtr[0] = ((unsigned long long)gdt << 16) | (sizeof(gdt) - 1);
gdtr[1] = ((unsigned long long)gdt >> 48);
asm volatile ("lgdt gdtr");
```

4.1.3 レジスタ CS・DS・SS の設定

CS・DS・SS 等のセグメントレジスタには、「セグメントセレクタ」と呼ばれる値を設定するのですが、「セグメントセレクタ」は少なくとも本書で扱う範囲においては「GDT 先頭からのオフセット」だと考えて問題ありません[*6]。

[*4] セグメント同士は領域を被って定義することもできます

[*5] 上述の通りそもそも 64 ビットのモードではセグメンテーションは一般的に無効な挙動となり、ここでは一応の安全のために設定しているだけです。この"4GB"を気にする必要は無いかと思います。

[*6] 厳密には下位 3 ビットに属性値を格納し、それより上位に GDT のインデックスを指定するのですが、少なくとも本書の範囲内では下位 3 ビットは常に 0 のためセグメントセレクタは GDT 先頭からのオフ

DS(データセグメント)・SS(スタックセグメント) のレジスタ設定は、単に mov 命令を使用してレジスタへ値を格納するだけです。ただし CS(コードセグメント) 設定は、現在実行中の CS を切り替えることになるため、例外へ陥ることが無いよう、「予めスタックに設定値を push し ret(関数リターン命令) で設定」という方法で設定します (リスト 4.3)。

▼**リスト** 4.3　CS・DS・SS **の設定**

```
/* DS・SS の設定 */
asm volatile ("movw $2*8, %ax\n"
              "movw %ax, %ds\n"
              "movw %ax, %ss\n");

/* CS の設定 */
unsigned short selector = 8;
unsigned long long dummy;
asm volatile ("pushq %[selector];"
              "leaq ret_label(%%rip), %[dummy];"
              "pushq %[dummy];"
              "lretq;"
              "ret_label:"
              : [dummy]"=r"(dummy)
              : [selector]"m"(selector));
```

4.1.4　まとめ: gdt_init 関数の定義

以上をまとめ、x86.c へ gdt_init 関数をリスト 4.4 の様に追加します[*7]。

▼**リスト** 4.4　040_intr/x86.c

```
#include <x86.h>

/* 追加（ここから） */
/* GDT の定義 */
const unsigned long long gdt[] = {
        0x0000000000000000,  /* NULL descriptor */
        0x00af9a000000ffff,  /* base=0, limit=4GB, mode=code(r-x),kernel */
        0x00cf93000000ffff   /* base=0, limit=4GB, mode=data(rw-),kernel */
};
unsigned long long gdtr[2];
/* 追加（ここまで） */

inline unsigned char io_read(unsigned short addr)
{
        /* ・・・ 省略 ・・・ */
}
```

セットと同じです。

[*7] インラインアセンブラ内のマジックナンバーの定数化はサボりました。。

```
/* 追加（ここから） */
void gdt_init(void)
{
        /* GDTR の設定 */
        gdtr[0] = ((unsigned long long)gdt << 16) | (sizeof(gdt) - 1);
        gdtr[1] = ((unsigned long long)gdt >> 48);
        asm volatile ("lgdt gdtr");

        /* DS・SS の設定 */
        asm volatile ("movw $2*8, %ax\n"
                      "movw %ax, %ds\n"
                      "movw %ax, %ss\n");

        /* CS の設定 */
        unsigned short selector = SS_KERNEL_CODE;
        unsigned long long dummy;
        asm volatile ("pushq %[selector];"
                      "leaq ret_label(%%rip), %[dummy];"
                      "pushq %[dummy];"
                      "lretq;"
                      "ret_label:"
                      : [dummy]"=r"(dummy)
                      : [selector]"m"(selector));
}
/* 追加（ここまで） */
```

併せて、x86.h へ gdt_init 関数の宣言とセグメントセレクタの定数 (SS_KERNEL_CODE,SS_KERNEL_DATA) を追加します (リスト 4.5)。

▼**リスト** 4.5　040_intr/include/x86.h

```
#ifndef _X86_H_
#define _X86_H_

#define SS_KERNEL_CODE 0x0008  /* 追加 */
#define SS_KERNEL_DATA 0x0010  /* 追加 */

inline unsigned char io_read(unsigned short addr);
void gdt_init(void);    /* 追加 */

#endif
```

4.2　CPU: IDT 設定

IDT(Interrupt Descriptor Table) は、割り込みと例外のハンドラを管理する全 256 エントリ・各エントリ 16 バイトのテーブルです[*8]。割り込みも例外も同じテーブルで扱う都合上、x86 では割り込み番号 0〜20 番 (IDT のインデックス 0〜20) を例外、割り込み

[*8] "IDT"という名前ですが割り込み (Interrupt) も例外 (Exception) も管理します。

番号 32〜255 番 (IDT のインデックス 32〜255) を割り込みとしています[*9]。

各エントリにはハンドラのアドレスといくつかのパラメータを設定し、GDT 同様にメモリ上に決まったデータ構造で配置します。lidt 命令を使用して IDTR というレジスタへ IDT を設定することで、CPU は当該の例外や割り込みが発生した際に IDT に設定されたハンドラを呼び出してくれます。

まとめると、IDT 設定で行うことは以下の通りです。

1. ハンドラ作成
2. IDT の定義
3. レジスタ IDTR の設定 (lidt 命令)

4.2.1 ハンドラ作成

今ハンドリングしたいものはキーボード割り込み (正確には KBC からの割り込み) だけなので、それ以外の割り込み/例外は余計な処理を行わないよう無限ループに陥るように設定することにします。そのため、作成する割り込みハンドラは以下の 2 つです。

- キーボードハンドラ (kbc_handler): キーボード (KBC) 割り込み時の処理を行う
- それ以外のハンドラ (default_handler): 無限ループ

割り込みハンドラは"handler.s"というファイルへアセンブラで記述します[*10]。まず、default_handler を handler.s へ記述します (リスト 4.6)。

▼**リスト** 4.6　040_intr/handler.s(default_handler)

```
        .global default_handler
default_handler:
        jmp     default_handler
```

リスト 4.6 について、".global"はシンボルを他のファイルからも参照できるようにするための指定です。その後は、"default_handler"シンボルへジャンプすることで、"default_handler"へ来ると無限ループするようにしています。

続いて、kbc_handler を定義します (リスト 4.7)。

[*9] 6.15 EXCEPTION AND INTERRUPT REFERENCE - Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3 : System Programming Guide

[*10] 関数定義時に"___attribute___ ((interrupt))"を付けておくと、後述する割り込みハンドラに対応した関数をコンパイラが生成してくれるらしいです。しかし、私の使用している GCC バイナリでは無効になっているようで"'interrupt' attribute directive ignored"と言われてしまうので、本書ではオーソドックスにアセンブラで記述しています。

▼**リスト** 4.7　040_intr/handler.s(kbc_handler)

```
/* ・・・ 省略 ・・・ */

/* 追加（ここから） */
        .global kbc_handler
kbc_handler:
        push    %rax
        push    %rcx
        push    %rdx
        push    %rbx
        push    %rbp
        push    %rsi
        push    %rdi
        call    do_kbc_interrupt
        pop     %rdi
        pop     %rsi
        pop     %rbp
        pop     %rbx
        pop     %rdx
        pop     %rcx
        pop     %rax
        iretq
/* 追加（ここまで） */
```

default_handler は、呼び出されると無限ループに陥るためハンドラからリターンすることが無いので、リスト 4.6 の様な記述で良いのですが、kbc_handler は必要な処理を終えたらユーザ空間へリターンする必要があります。リスト 4.7 では、割り込みハンドラとして必要な入口/出口処理のみアセンブラで記述し、主要な処理は C 言語の関数 (do_kbc_interrupt(後述)) を呼び出す (call 命令) ようにしています。

kbc_handler は以下を行っています。

- push/pop 命令
  - 汎用レジスタのスタックへの退避 (push) と復帰 (pop)[*11]
- call 命令
  - do_kbc_interrupt 関数呼び出し
- iretq 命令
  - 割り込みハンドラからのリターン処理

そして、KBC 割り込みの処理の本体である do_kbc_interrupt 関数は、kbc.c へリスト 4.8 の様に追加します。

[*11] 割り込みは、汎用レジスタを使用した何らかの処理の最中でも発生する可能性があります。そのため、割り込みハンドラ内で汎用レジスタを使用した結果、元の処理に戻ってきた時に汎用レジスタの内容が割り込み直前と異なってしまう事が無いよう、割り込みハンドラの入口で汎用レジスタをスタックへ退避し、出口でスタックから復帰しています。

▼**リスト** 4.8　040_intr/kbc.c(do_kbc_interrupt())

```
#include <kbc.h>
#include <x86.h>
#include <fbcon.h>      /* 追加 */

/* ・・・省略・・・ */

char getc(void)
{
        return keymap[get_keycode()];
}

/* 追加（ここから） */
void do_kbc_interrupt(void)
{
        /* ステータスレジスタの OBF がセットされていなければ return */
        if (!(io_read(KBC_STATUS_ADDR) & KBC_STATUS_BIT_OBF))
                return;

        /* make 状態でなければ return */
        unsigned char keycode = io_read(KBC_DATA_ADDR);
        if (keycode & KBC_DATA_BIT_IS_BRAKE)
                return;

        /* エコーバック処理 */
        char c = keymap[keycode];
        if (('a' <= c) && (c <= 'z'))
                c = c - 'a' + 'A';
        else if (c == '\n')
                putc('\r');
        putc(c);
}
/* 追加（ここまで） */
```

KBC 割り込み時、押下されたキーの情報を取得し、エコーバック処理を行うよう、前章で扱ったエコーバック処理を do_kbc_interrupt 関数へ持ってきています。[*12]

[*12] 一般的には割り込みハンドラで本格的な処理を行うべきでは無いです。(割り込みはシステム全体を止めるため) しかし、ここでは気にせず、割り込み契機で行いたい事は全てハンドラに実装してみることにします。(今はまだスケジューラも無く、ハンドラの処理が長引くことで被害を受ける人も居ないので。)

call 命令実行時、スタックポインタは 16 バイトアライメントされていること

64 ビットモードでの動作時、一部の CPU 命令にはスタックポインタ (RSP) が 16 の倍数になっていること (16 バイトアライメント) を前提にコンパイラが機械語コードを生成するものがあります。そのため、64 ビットモードにおいて、コンパイラは、関数実行開始時の RSP が 16 バイトアライメントされるように機械語コードを生成します。

C 言語の中で関数を呼び出したり呼びだされたりしている間はコンパイラがよしなに調節してくれるので 16 バイトアライメントを気にする必要はありません。しかし、今回の割り込みハンドラの様に自力でアセンブラを書いて関数呼び出しを行う場合は注意が必要です。

ただし、割り込み発生直後は、CPU が RSP を 16 バイトアライメント調節してから割り込みハンドラを呼び出してくれるので気にする必要はありません。

注意しなければならないのは割り込みハンドラ等のように自身で書いたアセンブラから call 命令で別の関数を呼び出す場合です。

call 命令は、1) 戻り先アドレス (8 バイト) をスタックに積む (RSP は-8 される)、2) 呼び出し先の関数へジャンプ、という振る舞いをします。そのため、割り込みハンドラ先頭では、CPU が調整してくれる事もあり、16 バイトアライメントであったとしても、call 命令の挙動により戻り先アドレスをスタックに積むため、呼び出し先の関数先頭時点で RSP は戻り先アドレス分の 8 を引いた値となり、16 バイトアライメントは崩れてしまいます。

本書では汎用レジスタ退避も兼ねて 7 個のレジスタ値 (各 8 バイト、計 56 バイト) をスタックに積んでから call 命令を実行しています。そのため、呼び出し先の do_kbc_interrupt 関数の先頭での RSP は、割り込みハンドラ先頭時点から合計 64(56 バイト + 8 バイト) 引いた値となり 16 バイトアライメントは保たれています。

なお、このコラムは以下の記事で勉強させてもらったものの受け売りです。詳しくは以下をご覧ください。

- x86-64 モードのプログラミングではスタックのアライメントに気を付けよう - uchan note
  - `http://uchan.hateblo.jp/entry/2018/02/16/232029`

4.2.2 IDT の定義

まず、IDT の各デスクリプタのデータ構造を"interrupt_descriptor"という名前の構造体として x86.h に用意しておきます (リスト 4.9)。

▼**リスト** 4.9　040_intr/include/x86.h(interrupt_descriptor **構造体**)

```
/* ・・・ 省略 ・・・ */
#define SS_KERNEL_DATA 0x0010

/* 追加（ここから） */
#define MAX_INTR_NO    256

struct interrupt_descriptor {
        unsigned short offset_00_15;
        unsigned short segment_selector;
        unsigned short ist      : 3;
        unsigned short _zero1   : 5;
        unsigned short type     : 4;
        unsigned short _zero2   : 1;
        unsigned short dpl      : 2;
        unsigned short p        : 1;
        unsigned short offset_31_16;
        unsigned int   offset_63_32;
        unsigned int   _reserved;
};
/* 追加（ここまで） */

inline unsigned char io_read(unsigned short addr);
/* ・・・ 省略 ・・・ */
```

次に、IDT 初期化関数 (intr_init) と IDT の各デスクリプタを設定する関数 (set_intr_desc) を"intr.c"というソースファイルへ実装します。それぞれの関数では以下を行うことにします。

- intr_init 関数: IDT の全エントリを default_handler で初期化
- set_intr_desc 関数: 引数で指定されたハンドラと割り込み番号でデスクリプタを設定する

intr.c はリスト 4.10 の通りです。

▼**リスト** 4.10　040_intr/intr.c(intr_init() **と** set_intr_desc())

```
#include <intr.h>
#include <x86.h>

#define DESC_TYPE_INTR 14
```

```
struct interrupt_descriptor idt[MAX_INTR_NO];

void default_handler(void);

void set_intr_desc(unsigned char intr_no, void *handler)
{
        idt[intr_no].offset_00_15 = (unsigned long long)handler;
        idt[intr_no].segment_selector = SS_KERNEL_CODE;
        idt[intr_no].type = DESC_TYPE_INTR;
        idt[intr_no].p = 1;
        idt[intr_no].offset_31_16 = (unsigned long long)handler >> 16;
        idt[intr_no].offset_63_32 = (unsigned long long)handler >> 32;
}

void intr_init(void)
{
        int i;
        for (i = 0; i < MAX_INTR_NO; i++)
                set_intr_desc(i, default_handler);
}
```

リスト 4.10 について、set_intr_desc 関数では、interrupt_descriptor 構造体の配列として intr.c 冒頭で定義した idt へ設定を行っています。どういったパラメータを設定する必要があるかは見たとおりですが、補足として、"type"はデスクリプタタイプで割り込み/例外の場合は 14 番を指定します。また、"p"は"Present"で、デスクリプタの存在フラグです[*13]。

併せて、intr.h を作成し、set_intr_desc 関数と intr_init 関数の宣言を追加します (リスト 4.11)。

▼**リスト** 4.11　040_intr/include/intr.h(set_intr_desc() と intr_init())

```
#ifndef _INTR_H_
#define _INTR_H_

void set_intr_desc(unsigned char intr_no, void *handler);
void intr_init(void);

#endif
```

また、kbc.c へ IDT 設定処理を kbc_init 関数として追加します (リスト 4.12)。

▼**リスト** 4.12　040_intr/kbc.c(kbc_init())

[*13] p フラグを 0 に設定しておくと、そのデスクリプタへのアクセスを例外 (割り込み番号 11 番、Segment Not Present) で検出できます。

```
#include <kbc.h>
#include <x86.h>
#include <fbcon.h>
#include <intr.h>       /* 追加 */

#define KBC_DATA_ADDR          0x0060
#define KBC_DATA_BIT_IS_BRAKE  0x80
#define KBC_STATUS_ADDR                0x0064
#define KBC_STATUS_BIT_OBF     0x01
#define KBC_INTR_NO            33      /* 追加 */

/* ・・・ 省略 ・・・ */

const char keymap[] = {
        /* ・・・ 省略 ・・・ */
};

void kbc_handler(void); /* 追加 */

/* ・・・ 省略 ・・・ */

void do_kbc_interrupt(void)
{
        /* ・・・ 省略 ・・・ */
}

/* 追加（ここから） */
void kbc_init(void)
{
        set_intr_desc(KBC_INTR_NO, kbc_handler);
}
/* 追加（ここまで） */
```

併せて、kbc.h へプロトタイプ宣言を追加します (リスト 4.13)。

▼**リスト** 4.13　040_intr/include/kbc.h

```
/* ・・・ 省略 ・・・ */

char getc(void);
void kbc_init(void);    /* 追加 */

#endif
```

4.2.3　レジスタ IDTR の設定 (lidt 命令)

続いて、lidt 命令を使用して IDT の先頭アドレスとサイズをレジスタ"IDTR(IDT Register)"へ設定します。

これらの処理は intr.c の intr_init 関数へ追加することにします (リスト 4.14)。

▼**リスト** 4.14　040_intr/intr.c(IDTR **の設定**)

```
/* ・・・ 省略 ・・・ */

struct interrupt_descriptor idt[MAX_INTR_NO];
unsigned long long idtr[2];     /* 追加 */

/* ・・・ 省略 ・・・ */

void intr_init(void)
{
        int i;
        for (i = 0; i < MAX_INTR_NO; i++)
                set_intr_desc(i, default_handler);

        /* 追加（ここから） */
        idtr[0] = ((unsigned long long)idt << 16) | (sizeof(idt) - 1);
        idtr[1] = ((unsigned long long)idt >> 48);
        __asm__ ("lidt idtr");
        /* 追加（ここまで） */
}
```

ここまでで IDT の設定も完了で、セグメンテーション含めて CPU の割り込み有効化のための設定は完了です。

4.3 CPU: 割り込み有効化 (sti 命令)

CPU 側の設定の最後に、sti(Set Interrupt Flag) 命令で割り込みの有効化を行います。"enable_cpu_intr"関数として x86.c へ追加します (リスト 4.15)。

▼**リスト** 4.15　040_intr/x86.c

```
/* ・・・ 省略 ・・・ */

/* 追加（ここから） */
inline void enable_cpu_intr(void)
{
        asm volatile ("sti");
}
/* 追加（ここまで） */

inline unsigned char io_read(unsigned short addr)
/* ・・・ 省略 ・・・ */
```

併せて、x86.h へプロトタイプ宣言を追加します (リスト 4.16)。

▼**リスト** 4.16　040_intr/include/x86.h

```
/* ・・・ 省略 ・・・ */

inline void enable_cpu_intr(void);      /* 追加 */
inline unsigned char io_read(unsigned short addr);
/* ・・・ 省略 ・・・ */
```

enable_cpu_intr 関数は main.c から呼び出しますが、main.c の変更はこの章の最後で他の変更と合わせて行います。

4.4 PIC: 割り込みコントローラ (PIC) 設定

最後は PIC の設定です。in/out の命令で PIC のレジスタへアクセスし、設定を行います。

やることは以下の 2 つです。

1. PIC の初期化処理を実施
2. 使用する割り込みのみ有効化

4.4.1 PIC 初期化処理の流れと使用するレジスタについて

PIC にはマスタ PIC とスレーブ PIC の 2 つがあり、初期化の際はマスタ/スレーブそれぞれに対して以下の初期化処理を実施します。

1. ICW(Initialize Control Word)1 へ ICW4 設定の要不要を設定
2. ICW2 へ割り込み番号のベース値設定
3. ICW3 へマスター時は 0x04 を、スレーブ時は 0x02 を設定
4. ICW4 へ各種パラメータ設定
5. OCW(Operation Control Word)1 へ割り込みマスク設定

また、上記レジスタの IO アドレスと書き込む値の意味は表 4.1 の通りです。

表 4.1 について、bit4 が 1 の書き込みが 0x0020(スレーブの場合 0x00a0) に対して行われると、それは ICW1 への書き込みとして扱われます。ICW1 へ書き込みが行われると PIC は初期化モードへ入ります。初期化モード後、0x0021(スレーブの場合 0x00a1) へ書き込みを行うと、書き込んだ値は ICW2、ICW3、ICW4(ICW1 の bit0 へ 1 を書き込んだ場合) の順で設定されます。最後の ICW(3 あるいは 4) への書き込みで初期化モードを抜け、以降の 0x0021(スレーブの場合 0x00a1) への書き込みは OCW1 への書き込みとして扱われます。

▼表 4.1　PIC のレジスタの IO アドレスと書き込む値の意味

IO アドレス (マスタ/スレーブ)	レジスタ名と書き込む値の意味
0x0020/0x00a0	ICW1
	bit7-4: 0x1
bit3-1: 0b000	
	bit0：ICW4 要不要 (必要=1 / 不要=0)
0x0021/0x00a1	ICW2
	bit7-3: 割り込み番号 [7:3] の値
	bit2-0: 0
0x0021/0x00a1	ICW3
	bit7-0: マスタは 0x04 を、スレーブは 0x02 を設定
0x0021/0x00a1	ICW4
	bit7-5: 0b000
	bit4：SFNM(特殊完全ネストモード=1 / ノーマル完全ネストモード=0)
	bit3-2: 0b00
	bit1：AEOI(自動 EOI モード=1 / ノーマル EOI モード=0)
	bit0：1
0x0021/0x00a1	OCW1
	bit7-0: 割り込みマスク (禁止=1 / 許可=0)

表 4.1 で重要な設定は、ICW2 で設定する割り込み番号の bit7〜bit3 の値です。マスタ・スレーブの PIC はそれぞれ、8 つずつ割り込みを受け付けることができます。それぞれの PIC では IR0〜IR7 という IR 番号に割り込みが対応しており、IR 番号は 3 ビットで表現できます。CPU へ割り込み番号として 8 ビットの値を通知する際に余った上位 5 ビットに好きな値を設定できるのがこの ICW2 です。

x86 CPU では外部割り込みは 32 番以降のため、ICW2 に 32(0x20) を設定しておけば、PIC は 32 番以降の値で割り込みを通知してくれます。なお、スレーブ側は 8 個分ずらして 40(0x28) を ICW2 へ設定します。

表 4.1 の ICW4 の AEOI(自動 EOI) や SFNM(特殊完全ネストモード) 等の機能は簡単のため本書では使用しません[*14]。それぞれが何であるかについては後述のコラムで説明していますので興味があれば見てみてください。ただし、"EOI"については別途割り込みハンドラを対応させる必要があるため、後述します。

なお、表 4.1 では現代の PC では固定値となるパラメータ等の説明は省略しています。PIC は十分に枯れたデバイスですので、詳しくはインターネット上で調べてみるか、参考

[*14] 高度な制御を行うなら今や APIC を勉強スベきかと思うので、本書では詳しく PIC に突っ込んでは行きません。

情報に挙げている書籍等を見てみてください。

4.4.2 PIC 初期化処理の実装

それでは、初期化処理を実装していきます。

IO アドレスへの書き込みの関数が無いので、まずは書き込みの関数を"io_write"という名前で x86.c へ追加します (リスト 4.17)。

▼**リスト** 4.17　040_intr/x86.c(io_write())

```
/* ・・・ 省略 ・・・ */
inline unsigned char io_read(unsigned short addr)
{
        /* ・・・ 省略 ・・・ */
}

/* 追加（ここから） */
inline void io_write(unsigned short addr, unsigned char value)
{
        asm volatile ("outb %[value], %[addr]"
                      :: [value]"a"(value), [addr]"d"(addr));
}
/* 追加（ここまで） */
/* ・・・ 省略 ・・・ */
```

併せて、x86.h へ io_write 関数の宣言を追加します (リスト 4.18)。

▼**リスト** 4.18　040_intr/include/x86.h

```
/* ・・・ 省略 ・・・ */
inline unsigned char io_read(unsigned short addr);
inline void io_write(unsigned short addr, unsigned char value); /* 追加 */
/* ・・・ 省略 ・・・ */
```

PIC の初期化処理は pic.c というソースファイルを作成し、そこへ pic_init 関数として実装することにします (リスト 4.19)。

▼**リスト** 4.19　040_intr/pic.c(pic_init())

```
#include <pic.h>
#include <x86.h>

#define MPIC_ICW1_ADDR 0x0020
#define MPIC_ICW2_ADDR 0x0021
#define MPIC_ICW3_ADDR 0x0021
#define MPIC_ICW4_ADDR 0x0021
#define MPIC_OCW1_ADDR 0x0021
```

```
#define SPIC_ICW1_ADDR 0x00a0
#define SPIC_ICW2_ADDR 0x00a1
#define SPIC_ICW3_ADDR 0x00a1
#define SPIC_ICW4_ADDR 0x00a1
#define SPIC_OCW1_ADDR 0x00a1

#define INTR_NO_BASE_MASTER    32
#define INTR_NO_BASE_SLAVE     40

void pic_init(void)
{
        /* マスタ PIC の初期化 */
        io_write(MPIC_ICW1_ADDR, 0x11);
        io_write(MPIC_ICW2_ADDR, INTR_NO_BASE_MASTER);
        io_write(MPIC_ICW3_ADDR, 0x04);
        io_write(MPIC_ICW4_ADDR, 0x01);
        io_write(MPIC_OCW1_ADDR, 0xff);

        /* スレーブ PIC の初期化 */
        io_write(SPIC_ICW1_ADDR, 0x11);
        io_write(SPIC_ICW2_ADDR, INTR_NO_BASE_SLAVE);
        io_write(SPIC_ICW3_ADDR, 0x02);
        io_write(SPIC_ICW4_ADDR, 0x01);
        io_write(SPIC_OCW1_ADDR, 0xff);
}
```

リスト 4.19 で設定している内容は先述した通りです。初期化完了後、OCW1 へマスタ・スレーブ共に全ての割り込みを無効にするようマスクしています。(次節で割り込み有効化関数を追加します。)

併せて、リスト 4.20 の様に pic.h を作成します。

▼**リスト** 4.20　040_intr/include/pic.h(pic_init())

```
#ifndef _PIC_H_
#define _PIC_H_

void pic_init(void);

#endif
```

4.4.3　割り込み有効化関数の実装

pic_init 関数では全ての割り込みをマスクしましたので、pic.c は割り込み有効化関数を提供し、デバイス側から割り込みマスクの解除が行えるようにします。

マスクの特定のビットのみ操作するためには現在のマスク値を読み出せる必要があります。現在の割り込みマスク値は IMR(Interrupt Mask Register) を読み出すことで確認できます (表 4.2)。

▼表 4.2　IMR のアドレスと読み出す値の意味

IO アドレス (マスタ/スレーブ)	レジスタ名と読み出す値の意味
0x0021/0x00a1	IMR
	bit7-0: 割り込みマスク (禁止=1 / 許可=0)

表 4.2 について、IMR のアドレスは OCW1 と同じです。読めば IMR としてマスク値が読み出せ、書けば OCW1 としてマスク値を設定できます。

それでは、pic.c へ enable_pic_intr 関数として割り込み有効化関数を追加します (リスト 4.21)。

▼**リスト** 4.21　040_intr/pic.c(enable_pic_intr())

```
/* ・・・ 省略 ・・・ */
#define MPIC_ICW4_ADDR 0x0021
#define MPIC_OCW1_ADDR 0x0021
#define MPIC_IMR_ADDR  0x0021  /* 追加 */
/* ・・・ 省略 ・・・ */

void pic_init(void)
{
        /* ・・・ 省略 ・・・ */
}

/* 追加（ここから） */
void enable_pic_intr(unsigned char intr_no) /* ※ スレーブ未対応 */
{
        /* intr_no 番のビットのみ立っているビットフィールド（ir_bit）を作成 */
        unsigned char ir_no = intr_no - INTR_NO_BASE_MASTER;
        unsigned char ir_bit = 1U << ir_no;

        /* 現在のマスク値を取得 */
        unsigned char mask = io_read(MPIC_IMR_ADDR);

        /* 既にマスク解除済みならば何もせず return */
        if (!(ir_bit & mask))
                return;

        /* マスク解除 */
        io_write(MPIC_OCW1_ADDR, mask - ir_bit);
}
/* 追加（ここまで） */
```

リスト 4.21 では、スレーブ側を設定することは本書では無いので、簡単のためマスタ PIC にのみ対応しています。

併せて、pic.h へプロトタイプ宣言を追加します (リスト 4.22)。

▼**リスト** 4.22　040_intr/include/pic.h(enable_pic_intr())

```
#ifndef _PIC_H_
#define _PIC_H_

void pic_init(void);
void enable_pic_intr(unsigned char intr_no);    /* 追加 */

#endif
```

これで特定の割り込みの有効化が行えるようになったので、kbc.c の kbc_init 関数へ割り込み有効化の処理を追加します (リスト 4.23)。

▼**リスト** 4.23　040_intr/kbc.c(KBC **割り込み有効化**)

```
#include <kbc.h>
#include <x86.h>
#include <fbcon.h>
#include <intr.h>
#include <pic.h>        /* 追加 */

/* ・・・ 省略 ・・・ */

void kbc_init(void)
{
        set_intr_desc(KBC_INTR_NO, kbc_handler);
        enable_pic_intr(KBC_INTR_NO);   /* 追加 */
}
```

4.4.4　PIC の EOI(End Of Interrupt) 設定

"EOI"は"End Of Interrupt"の略で、CPU から PIC に対して割り込み処理の終了を通知するものです。PIC の OCW2 レジスタへ EOI 設定を行うと PIC は CPU が割り込み処理を終了したと判断し、次の割り込みを受け付けるようになります。OCW2 レジスタについては表 4.3 の通りです[*15]。

▼**表** 4.3　PIC **の** OCW2 **レジスタの** IO **アドレスと書き込む値の意味**

IO **アドレス** (**マスタ**/**スレーブ**)	**レジスタ名と書き込む値の意味**
0x0020/0x00a0	OCW2
	bit7-5: EOI 方法 (本書では「指定 EOI(0b011)」を使用)
	bit4-3: 0
	bit2-1: EOI 対象の IR 番号指定

[*15] OCW2 は ICW1 と同じアドレスですが、書き込まれた時の bit4-3 の値によって内部で判別されています。

「指定された割り込みの EOI を設定する」関数を用意します。"set_pic_eoi"という名前で pic.c へ追加します (リスト 4.24)。

▼**リスト** 4.24　040_intr/pic.c(set_pic_eoi())

```
#define MPIC_ICW1_ADDR 0x0020
#define MPIC_OCW2_ADDR 0x0020  /* 追加 */
/* ・・・ 省略 ・・・ */

#define PIC_OCW2_BIT_MANUAL_EOI        0x60    /* 追加 */

/* ・・・ 省略 ・・・ */

void enable_pic_intr(unsigned char intr_no)
{
        /* ・・・ 省略 ・・・ */
}

/* 追加（ここから） */
void set_pic_eoi(unsigned char intr_no) /* ※ スレーブ未対応 */
{
        /* IR 番号計算 */
        unsigned char ir_no = intr_no - INTR_NO_BASE_MASTER;

        /* ir_no 番に対して EOI 設定 */
        io_write(MPIC_OCW2_ADDR, PIC_OCW2_BIT_MANUAL_EOI | ir_no);
}
/* 追加（ここまで） */
```

併せて、pic.h へプロトタイプ宣言を追加します (リスト 4.25)。

▼**リスト** 4.25　040_intr/include/pic.h

```
#ifndef _PIC_H_
#define _PIC_H_

void pic_init(void);
void enable_pic_intr(unsigned char intr_no);
void set_pic_eoi(unsigned char intr_no);        /* 追加 */

#endif
```

set_pic_eoi 関数を使用して、kbc.c の do_kbc_interrupt 関数へ EOI 処理を追加します (リスト 4.26)。

▼**リスト** 4.26　040_intr/kbc.c(set_pic_eoi())

```
/* ・・・ 省略 ・・・ */
void do_kbc_interrupt(void)
{
        /* ステータスレジスタの OBF がセットされていなければ return */
```

```
        if (!(io_read(KBC_STATUS_ADDR) & KBC_STATUS_BIT_OBF))
                goto kbc_exit;  /* 変更 */

        /* make 状態でなければ return */
        unsigned char keycode = io_read(KBC_DATA_ADDR);
        if (keycode & KBC_DATA_BIT_IS_BRAKE)
                goto kbc_exit;  /* 変更 */

        /* エコーバック処理 */
        char c = keymap[keycode];
        if (('a' <= c) && (c <= 'z'))
                c = c - 'a' + 'A';
        else if (c == '\n')
                putc('\r');
        putc(c);

/* 追加（ここから） */
kbc_exit:
        /* PIC へ割り込み処理終了を通知（EOI） */
        set_pic_eoi(KBC_INTR_NO);
/* 追加（ここまで） */
}
/* ・・・ 省略 ・・・ */
```

AEOI(自動 EOI) モードとは

表 4.1 の ICW4 の bit1 で設定する"AEOI"は「自動 EOI」と呼ばれるもので、設定されているとソフトウェア的な割り込みハンドラの処理とは非同期に PIC が次の割り込みを受け付けるようになります。

本書では簡単のため、割り込みハンドラで EOI を設定してから次の割り込みを受け付けるようにするので、AEOI は使用しません。

SFNM(特殊完全ネストモード) とは

表 4.1 の ICW4 の bit4 の"SFNM"は「特殊完全ネストモード」と呼ばれるモードの有効/無効の設定です。スレーブから多重に割り込みを受け付けるモードを使用するか否かを設定します。

PIC のマスタ・スレーブの関係は、スレーブが割り込み要求を上げる信号線がマスターの割り込み要求を受け付けるピンに接続されている関係となっており、スレーブ側で受け付けた割り込みは CPU ではなくマスター PIC へ伝わります。

"AEOI"で前述した通り、PIC は割り込み処理中は次の割り込みを受け付けません。そのため、スレーブ側で優先度が低い割り込みが発生し、マスタ PIC を通して CPU が処理中の間、スレーブで優先度の高い割り込みが発生すると、マスタがそもそも割り込みを受け付けないため優先度が高いにも関わらず割り込みが受け付けられないという優先度の逆転現象が発生します。

このような問題を解決するためにスレーブから多重に割り込みを受け付ける事ができるようにする設定が SFNM(特殊完全ネストモード) です。

ただし、SFNM が有効な場合、スレーブから多重に割り込みを受け付けることになるため、CPU から EOI を発行する際、スレーブにペンディング中の割り込みがないかチェックする必要があります。

4.5 仕上げ: main.c と Makefile へ反映する

4.5.1 main.c と Makefile へこれまでの変更を盛り込む

以上で CPU 側と PIC 側それぞれの周辺のソースコードの追加・変更は完了です。
最後に main.c と Makefile を変更します。
main.c の変更はリスト 4.27 の通りです。

▼**リスト** 4.27　040_intr/main.c

```
#include <x86.h>        /* 追加 */
#include <intr.h>       /* 追加 */
#include <pic.h>        /* 追加 */
#include <fb.h>
#include <kbc.h>

void start_kernel(void *_t __attribute__ ((unused)), struct framebuffer *fb,
```

```
                    void *_fs_start __attribute__ ((unused)))
{
        /* フレームバッファ周りの初期化 */
        fb_init(fb);
        set_fg(255, 255, 255);
        set_bg(0, 70, 250);
        clear_screen();

        /* 変更（ここから） */
        /* CPU 周りの初期化 */
        gdt_init();
        intr_init();

        /* 周辺 IC の初期化 */
        pic_init();
        kbc_init();

        /* CPU の割り込み有効化 */
        enable_cpu_intr();

        /* 何もせず待つ */
        while (1);
        /* 変更（ここまで） */
}
```

エコーバック処理を KBC のハンドラへ持っていったので、start_kernel 関数は各種の初期化処理を終えると何もせず待つようになりました。

Makefile の変更はリスト 4.28 の通りです。

▼**リスト** 4.28　040_intr/Makefile

```
/* ・・・ 省略 ・・・ */
$(TARGET): main.o fbcon.o fb.o font.o kbc.o x86.o intr.o pic.o handler.o  # 変更
        ld $(LDFLAGS) -o $@ $+
/* ・・・ 省略 ・・・ */
%.o: %.c
        gcc $(CFLAGS) -c -o $@ $<
%.o: %.s                                # 追加
        gcc $(CFLAGS) -c -o $@ $<       # 追加
```

リスト 4.28 の「変更」の箇所で"intr.o"と"pic.o"と"handler.o"を追加しています。

4.5.2　halt で待つようにする

これでも良いのですが、「何もせず待つ」の無限ループがあまりエコではないです。

x86 CPU には hlt(halt) 命令があります。この命令を実行すると、命令実行を止めて CPU を HALT 状態に遷移させ、割り込み等で復帰します。「何もせず待つ」の無限ループでは hlt 命令を実行するようにしてみます。

まずは、x86.c へ"cpu_halt"という名前で hlt 命令を実行する関数を追加します (リス

ト 4.29)。

▼**リスト** 4.29　040_intr/x86.c

```
/* ・・・ 省略 ・・・ */

inline void enable_cpu_intr(void)
{
        asm volatile ("sti");
}

/* 追加（ここから） */
inline void cpu_halt(void)
{
        asm volatile ("hlt");
}
/* 追加（ここまで） */
/* ・・・ 省略 ・・・ */
```

併せて x86.h へプロトタイプ宣言を追加します (リスト 4.30)。

▼**リスト** 4.30　040_intr/include/x86.h(cpu_halt())

```
/* ・・・ 省略 ・・・ */
inline void enable_cpu_intr(void);
inline void cpu_halt(void);     /* 追加 */
/* ・・・ 省略 ・・・ */
```

そして、main.c を修正します (リスト 4.31)。

▼**リスト** 4.31　040_intr/main.c(cpu_halt())

```
/* ・・・ 省略 ・・・ */
void start_kernel(void *_t __attribute__ ((unused)), struct framebuffer *fb,
                  void *_fs_start __attribute__ ((unused)))
{
/* ・・・ 省略 ・・・ */

        /* halt して待つ */
        while (1)               /* 変更 */
                cpu_halt();     /* 追加 */
}
```

以上でキーボード入力の割り込み対応の追加・変更は終わりです。

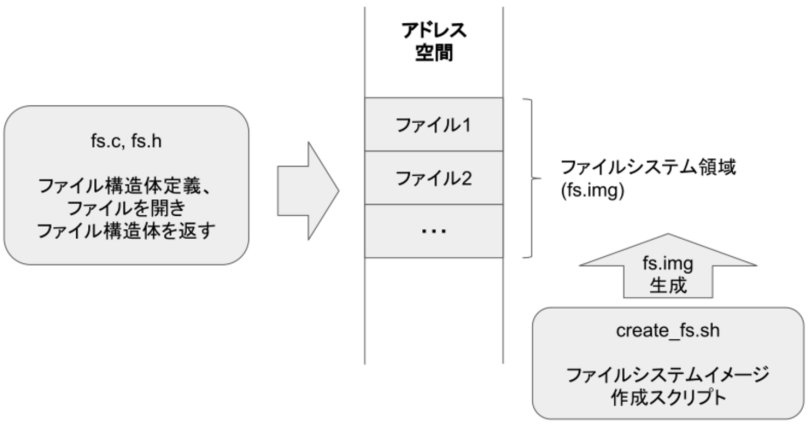
ポーリング版と今回の割り込み版は、見た目はどちらも同じですが、それぞれ実際に試してみると、キー入力を行っていない時は halt によって CPU が休んでいる様子が CPU ファンの音などで確認できるかと思います。(見た目変わらないのでスクリーンショットは省略します。)

第 5 章

メモリ上の簡易ファイルシステムを用意する

次の章で画像ビューアを作ってみるに当たり、メモリ上の簡易的なファイルシステムを用意してみます。

この章では図 5.1 の構成でファイルシステム関連の機能を実装します。



▲図 5.1　この章で作る機能の構成

poiboot が配置したファイルシステムを解釈するファイルシステム機能 (fs.c と fs.h) と、与えられたファイルからファイルシステムイメージを生成するスクリプト (create_fs.sh)

を作成します。

5.1 poiboot の第 2 のロード機能

poiboot は kernel.bin だけでなく、もうひとつイメージをロードしてくれる機能があります。"fs.img"というファイル名で kernel.bin と同じ階層に配置しておくと (リスト 5.1)、poiboot.conf の 2 行目に記載したアドレスへに配置します。

▼**リスト** 5.1　fs.img **は** kernel.img **と同じ階層へ配置**

```
fs ディレクトリ、あるいは USB フラッシュメモリ等
├── EFI/
│   └── BOOT/
│       └── BOOTX64.EFI
├── kernel.bin
└── fs.img  ← kernel.bin と同じ階層に配置
```

poiboot はカーネルへジャンプする際、第 3 引数にファイルシステムの先頭アドレスを渡してくれるので、カーネルはエントリ関数 (main.c の start_kernel 関数) の第 3 引数からファイルシステムの先頭アドレスを知ることができます。

この章では、ファイルシステム領域上のバイナリ列をファイルと見立てるルールを決めることで、簡易的なファイルシステムとします。

5.2 単一のテキストファイルを読んでみる

まずは poiboot の機能確認として、単一のテキストファイルを fs.img というファイル名で配置し、start_kernel 関数の第 3 引数で受け取ったアドレスへファイルの内容がロードされていることを確認してみます。

この節のサンプルディレクトリは"050_fs_read_text"です。これまで同様に文中では前回のサンプルを元に説明しますので、説明に沿って作業を行う場合は前回のサンプル"040_intr"を元に作業を行ってください。

まずは fs.img(テキストファイル) を用意します。

```
$ echo -n 'HELLO\0' > fs.img
```

できあがった"fs.img"を kernel.bin と同じディレクトリに配置してください。

次に、start_kernel 関数第 3 引数に指定されたアドレス (fs.img のロード先アドレス) を読むように kernel.bin のプログラムを改造します。main.c へリスト 5.2 のように追記

します。

▼リスト 5.2　050_fs_read_text/main.c

```
#include <x86.h>
#include <intr.h>
#include <pic.h>
#include <fb.h>
#include <kbc.h>
#include <fbcon.h>      /* 追加 */

void start_kernel(void *_t __attribute__ ((unused)), struct framebuffer *fb,
                  void *_fs_start)
{
        /* ・・・ 省略 ・・・ */

        /* CPU の割り込み有効化 */
        enable_cpu_intr();

        /* fs.img(テキストファイル) を読む */       /* 追加 */
        char *hello_str = (char *)_fs_start;    /* 追加 */
        puts(hello_str);                        /* 追加 */

        /* halt して待つ */
        while (1)
                cpu_halt();
```

fs.img が start_kernel 関数第 3 引数"_fs_start"へロードされることで、文字列"HELLO\0"が_fs_start に格納されたアドレスから並ぶ事になります。そのアドレスを char *として参照し、puts 関数で出力するだけです。

実行すると図 5.2 の様に表示されます。

▲図 5.2　050_fs_read_text の実行結果

なお、KBC 割り込み時のエコーバックは残っているのでキー入力すると"HELLO"に続いて表示されます。

また、"font.c"にアルファベットは大文字しか用意していないので fs.img として小文字のテキストを用意すると化けます。

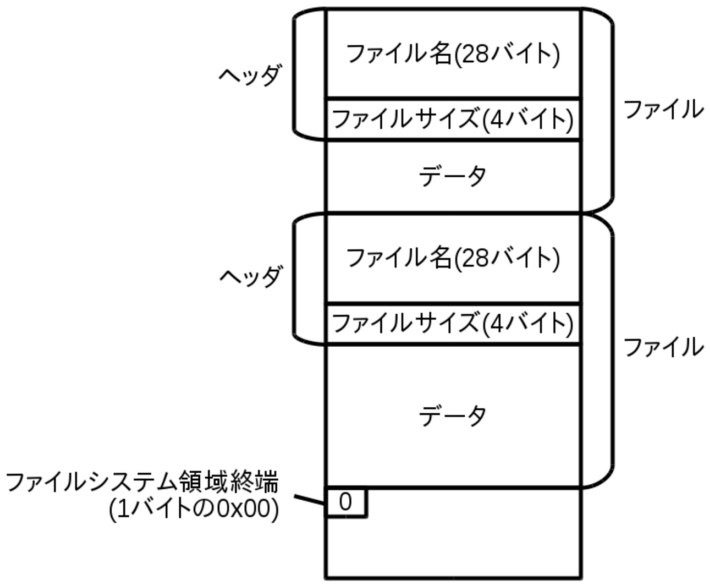
5.3 ファイルシステムを考える

前節では fs.img というファイル名で kernel.bin と同じ階層へ配置しておけば、poiboot が_fs_start のアドレスへロードされることを確認しました。

次は、単なるバイト列である"fs.img"の中に複数の「ファイル」を表現する「ルール」を考えます。本書で「ファイルシステム」として考える機能はこのような「ルール」です。シンプルに以下を考えてみました[*1]。

1. 各「ファイル」は「ヘッダ」と「データ」で構成される
2. 「ヘッダ」は「ファイル名 (28 バイト)」と「ファイルサイズ (4 バイト)」で構成される
3. fs.img には、ファイルが先頭から隙間なく並べられている
4. ファイルシステム領域の終わりは 1 バイトの 0x00 で示す

これらのルールを図示すると図 5.3 の通りとなります。



▲図 5.3　バイト列をファイルへ見立てるルール

[*1] ファイル名が ASCII で 28 文字までであったり、ファイルサイズが 4GB までだったりしますが、今必要な事以上のことは考えないことにします。

5.4 ファイルシステムを実装する

それでは、前節で説明したファイルシステムを解釈できるように処理を追加していきます。

この節のサンプルディレクトリは"051_fs_read_fs"です。この節も一つ前のサンプル"050_fs_read_text"を元に作ります。

ファイルシステムに関しては"fs.h"と"fs.c"へ記述することにします。

まず、"fs.h"を作成します (リスト 5.3)。fs.h では以下を行います。

- ファイル構造体 (struct file) の定義
- fs_init 関数のプロトタイプ宣言
- open 関数のプロトタイプ宣言

▼**リスト** 5.3　051_fs_read_fs/include/fs.h

```
#ifndef _FS_H_
#define _FS_H_

#define FILE_NAME_LEN  28

struct file {
        char name[FILE_NAME_LEN];
        unsigned int size;
        unsigned char data[0];
};

void fs_init(void *_fs_start);
struct file *open(char *name);

#endif
```

次に、"fs.c"へ各関数の実装を行っていきたいところですが、先に"common.c"と"common.h"というソースファイルを用意して「文字列比較関数 (strcmp)」と「ヌルポインタ (NULL)」を定義しておきます。

"common.c"の内容はリスト 5.4 の通りです。

▼**リスト** 5.4　051_fs_read_fs/common.c

```
#include <common.h>

int strcmp(char *s1, char *s2)
{
        char is_equal = 1;
```

```
        for (; (*s1 != '\0') && (*s2 != '\0'); s1++, s2++) {
                if (*s1 != *s2) {
                        is_equal = 0;
                        break;
                }
        }

        if (is_equal) {
                if (*s1 != '\0') {
                        return 1;
                } else if (*s2 != '\0') {
                        return -1;
                } else {
                        return 0;
                }
        } else {
                return (int)(*s1 - *s2);
        }
}
```

strcmp 関数は、文字列比較ができればどのような実装でも構いません。
"common.h"の内容はリスト 5.5 の通りです。

▼**リスト** 5.5　051_fs_read_fs/include/common.h

```
#ifndef _COMMON_H_
#define _COMMON_H_

#define NULL   (void *)0

int strcmp(char *s1, char *s2);

#endif
```

これらを使用して、fs.c へ fs_init 関数と open 関数を定義します (リスト 5.6)。それぞれの関数が行うことは以下の通りです。

- fs_init 関数
  - 引数で渡されたポインタを fs.c 内のファイルシステムスタートアドレス (struct file *fs_start) へ設定
- open 関数
  - ファイル名を渡すとファイルシステム上の対応する struct file *を返す

▼**リスト** 5.6　051_fs_read_fs/fs.c

```
#include <fs.h>
#include <common.h>

#define END_OF_FS      0x00

struct file *fs_start;

void fs_init(void *_fs_start)
{
        fs_start = _fs_start;
}

struct file *open(char *name)
{
        struct file *f = fs_start;
        while (f->name[0] != END_OF_FS) {
                if (!strcmp(f->name, name))
                        return f;

                f = (struct file *)((unsigned long long)f + sizeof(struct file)
                                    + f->size);
        }

        return NULL;
}
```

open 関数は、ファイルシステムの終わりまで while ループを回し、引数で与えたファイル名と同じものが見つかれば struct file のポインタを返しています (見つからなければ NULL を返します)。

そして、追加した関数を使用して複数のテキストファイル"HELLO.TXT"と"FOO.TXT"を読む処理を main.c へ追加してみます (リスト 5.7)。

▼**リスト** 5.7　051_fs_read_fs/main.c

```
/* ・・・ 省略 ・・・ */
#include <fbcon.h>
#include <fs.h> /* 追加 */

void start_kernel(void *_t __attribute__ ((unused)), struct framebuffer *fb)
{
        /* ・・・ 省略 ・・・ */

        /* ファイルシステムの初期化 */      /* 追加 */
        fs_init(_fs_start);             /* 追加 */

        /* CPU の割り込み有効化 */
        enable_cpu_intr();

        /* 変更（ここから） */
        /* HELLO.TXT と FOO.TXT を 1 行目のみ読む */
        struct file *hello = open("HELLO.TXT");
        if (hello) {
```

```
                puts((char *)hello->name);
                putc(' ');
                puts((char *)hello->data);
        } else {
                puts("HELLO.TXT IS NOT FOUND.");
        }
        puts("\r\n");

        struct file *foo = open("FOO.TXT");
        if (foo) {
                puts((char *)foo->name);
                putc(' ');
                puts((char *)foo->data);
        } else {
                puts("FOO.TXT IS NOT FOUND.");
        }
        puts("\r\n");
        /* 変更（ここまで） */

        /* halt して待つ */
        while (1)
                cpu_halt();
}
```

リスト 5.7 では、open 関数を使用して、ファイルシステムからファイル名に対応した struct file *を取得し、ファイル名とデータの表示を行っています。データは先頭アドレスをそのまま puts 関数へ渡しているだけなので、1 行目だけ表示されます。

最後に Makefile を変更します (リスト 5.8)。

▼**リスト** 5.8　051_fs_read_fs/Makefile

```
TARGET = kernel.bin
CFLAGS = -Wall -Wextra -nostdinc -nostdlib -fno-builtin -fno-common -Iinclude
LDFLAGS = -Map kernel.map -s -x -T kernel.ld
OBJS = main.o fbcon.o fb.o font.o kbc.o x86.o intr.o pic.o handler.o    \ # 追加
        fs.o common.o                                                    # 追加

$(TARGET): $(OBJS)      # 変更
        ld $(LDFLAGS) -o $@ $+

# ・・・ 省略 ・・・
```

"$(TARGET)"の依存リストが長くなってきたので、$(OBJS) という変数へ分けました。

実行してみるには、あとファイルシステムイメージ (fs.img) の生成が必要です。次節で作成します。

5.5 ファイルシステムイメージ (fs.img) 出力スクリプトを作る

カーネル側へのファイルシステム機能の追加は終わりましたので、最後にファイルシステムイメージ (fs.img) を生成します。

サンプルディレクトリは"052_fs_create_fs"です。

「ファイルシステムを考える」の節で説明したルールを満たしたバイナリを"fs.img"として生成できれば何でも良いです。

ここではシェルスクリプトで実装してみました。内容はリスト 5.9 の通りです。

▼**リスト** 5.9　052_fs_create_fs/create_fs.sh

```
#!/bin/bash

FILE_NAME_LEN=28
FS_IMG_NAME='fs.img'

while [ -n "$1" ]; do
        # ヘッダ作成
        ## ファイル名を出力
        name=$(basename $1)
        echo -ne "$name\0"
        dd if=/dev/zero count=$((${FILE_NAME_LEN} - ${#name} - 1)) bs=1 \
           status=none

        ## ファイルサイズを出力
        ### ファイルサイズを 16 進 8 桁へ変換
        size=$(stat -c '%s' $1)
        size_hex=$(printf "%08X" $size)
        ### リトルエンディアンへ変換
        size_hex_le=''
        for b in $(echo $size_hex | fold -w2 -b); do
                size_hex_le="$b$size_hex_le"
        done
        ### 8 ビットずつ出力
        for b in $(echo $size_hex_le | fold -w2 -b); do
                printf '%b' "\x$b"
        done

        # データ出力
        dd if=$1 status=none

        # 位置変数をシフト
        shift
done > ${FS_IMG_NAME}        # while ループ内の出力は fs.img へ出力

# ファイルシステムの終わり (0x00) を fs.img へ追記
echo -ne "\x00" >> ${FS_IMG_NAME}
```

スクリプトの内容についてはコメントに記載のとおりです。

スクリプトは以下のように、ファイルシステムへ格納したいファイルを引数に指定して実行します。

```
$ ./create_fs.sh ファイル1 ファイル2 ...
$ ls fs.img
fs.img  # fs.img が生成される
```

ここでは、以下のように 2 つのファイルを作成し、それらのファイルを含む fs.img を作成してみます。

```
$ echo -n "HELLO\0" > HELLO.TXT            # HELLO.TXT を作成
$ hexdump -C HELLO.TXT                     # 中身を確認
00000000  48 45 4c 4c 4f 00                                       |HELLO.|
00000006
$ echo -n "FOO\0" > FOO.TXT                # FOO.TXT を作成
$ hexdump -C FOO.TXT                       # 中身を確認
00000000  46 4f 4f 00                                             |FOO.|
00000004
$ ./create_fs.sh HELLO.TXT FOO.TXT # fs.img を作成
$ ls fs.img                                # 作成されたことを確認
fs.img
```

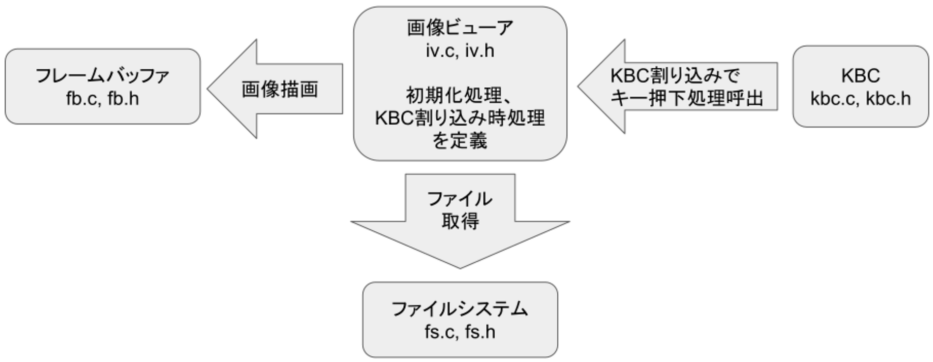
前節の変更を適用して作成した kernel.bin とここで作成した fs.img を実行すると図 5.4 の様に 2 つのファイルのファイル名とファイルの内容が表示されます。

▲図 5.4　052_fs_create_fs の実行結果

第 6 章

画像ビューアアプリを作ってみる

「アプリ」といってもバイナリレベルでカーネルと分けたり、動作時の権限を変えたりは本書ではしません。これまでプラットフォームとして実装してきた「フレームバッファ」・「キーボードコントローラ」・「割り込み」・「ファイルシステム」等の道具を使用して何らかの用途を実現するサンプルとして、本書では画像のビューアを作ってみます。作成する画像ビューアの構成は図 6.1 の通りです。



▲図 6.1　画像ビューアの構成

6.1 仕様

仕様は以下の通りとします。

- 起動するとファイルシステム内の順序で 1 番目の画像を表示する
- 'j' キー押下で次の画像、'k' キー押下で前の画像を表示する
- 表示する順序はファイルシステム内の並び順
- ファイルシステム内には画像ファイルしか無いものとする
- 画像ファイルのフォーマットは BGRA 各 8 ビット (フレームバッファのピクセルフォーマット)
- 画像ファイルのサイズ (幅・高さ) はフレームバッファ全画面表示の解像度

6.2 画像ファイルを用意する

画像ファイルの仕様は前述の通り以下です。

- フォーマット: BGRA(各色 8 ビット)
- サイズ (幅・高さ): フレームバッファ全画面表示の解像度

なお、この中で、「サイズ (幅・高さ)」は具体的な値がまだわかっていませんので見てみます。

6.2.1 putd 関数を追加し、フレームバッファ解像度を画面表示して確認する

フレームバッファの解像度は struct framebuffer 構造体中に存在します。この値を表示させて確認したいところですが、そういえばこれまで、数値を画面へ表示する機能は実装していませんでした。

ここでは数値を表示する関数を putd(put decimal) という名前で追加し、フレームバッファの解像度を画面へ表示してみます。

この節のサンプルディレクトリは"060_iv_show_fb_resol"です。前章の"051_fs_read_fs"を元に作ります。

それでは、putd 関数を fbcon.c へ追加してみます (リスト 6.1)。

▼**リスト** 6.1　060_iv_show_fb_resol/fbcon.c

```
#include <fbcon.h>
#include <fb.h>
#include <font.h>

/* 追加（ここから） */
/* 64bit unsigned の最大値 0xffffffffffffffff は
 * 10 進で 18446744073709551615(20 桁) なので'\0' 含め 21 文字分のバッファで足りる */
#define MAX_STR_BUF    21
/* 追加（ここまで） */

/* ・・・ 省略 ・・・ */

void puts(char *s)
{
        while (*s != '\0')
                putc(*s++);
}

/* 追加（ここから） */
void putd(unsigned long long val, unsigned char num_digits)
{
        char str[MAX_STR_BUF];

        int i;
        for (i = num_digits - 1; i >= 0; i--) {
                int digit = val % 10;
                str[i] = '0' + digit;
                val /= 10;
        }
        str[num_digits] = '\0';

        puts(str);
}
/* 追加（ここまで） */
```

併せて、fbcon.h へ putd 関数のプロトタイプ宣言を追加します。

▼**リスト** 6.2　060_iv_show_fb_resol/include/fbcon.h

```
/* ・・・ 省略 ・・・ */
void puts(char *s);
void putd(unsigned long long val, unsigned char num_digits);    /* 追加 */

#endif
```

putd 関数は第 1 引数に表示したい値を、第 2 引数に表示桁数を指定します。

それでは、追加した putd 関数を使用してフレームバッファの解像度情報を表示してみます (リスト 6.3)。

▼**リスト** 6.3　060_iv_show_fb_resol

```
/* ・・・ 省略 ・・・ */

void start_kernel(void *_t __attribute__ ((unused)), struct framebuffer *_fb)
{
        /* フレームバッファ周りの初期化 */
        fb_init(_fb);
        /* ・・・ 省略 ・・・ */

        /* CPU の割り込み有効化 */
        enable_cpu_intr();

        /* 変更（ここから） */
        /* 水平解像度（Horizontal Resolution）と
         * 垂直解像度（Vertical Resolution）を表示 */
        puts("HR ");
        putd(fb.hr, 4);
        puts("\r\n");
        puts("VR ");
        putd(fb.vr, 4);
        puts("\r\n");
        /* 変更（ここまで） */

        /* halt して待つ */
        while (1)
                cpu_halt();
}
```

struct framebuffer の値の参照には start_kernel 関数の仮引数の"_fb"ではなく、fb_init 関数内で設定され、fb.h で extern されている struct framebuffer fb を使用しています[*1]。

QEMU で実行すると図 6.2 のように表示されました。

▲図 6.2　060_iv_show_fb_resol の実行結果 (QEMU)

[*1] start_kernel 関数内においてはどちらを用いても良いのですが、カーネル内の処理は fb.h のものを使うように統一しています。

解像度は 800x600 のようです。
ただし、実機 (Lenovo) で試した所、図 6.3 のように表示されました。

▲図 6.3　060_iv_show_fb_resol の実行結果 (実機)

poiboot がカーネルへ渡すフレームバッファの解像度は起動時の UEFI ファームウェアのデフォルトの解像度そのままです。
ちゃんと解像度設定を行うならば poiboot で行うのが UEFI の機能が使えるので楽かと思いますが[*2]、ここでは単に QEMU 側を採用することにします。
そのため、以降、フレームバッファの解像度は 800x600 だとし、サンプルプログラムは QEMU 上での動作を想定します。といっても解像度が違うだけなので、以降で紹介する画像表示プログラムを複数解像度対応させるか、ご自身のマシンの解像度を調べて 800x600 の箇所を書き替えていただければ実機でも動作させられるかと思います。

6.2.2　ImageMagick(convert コマンド) で画像を変換する

次に画像ファイルを変換します。
画像ファイルの変換には ImageMagick という画像編集のツール郡に含まれる"convert" コマンドを使用します。ImageMagick がインストールされていない場合は以下のコマンドでインストールを行ってください。

[*2] UEFI での解像度変更 (グラフィックモード変更) 方法は、前著「フルスクラッチで作る!UEFI ベアメタルプログラミング パート 2」の「1.4 テキストモードを変更する」のコラム「グラフィックモードの情報を確認し、変更するには」に記載しています。本書同様、PDF 版は無料でダウンロードできます。「参考情報」にリンクを記載しておりますので興味があればご覧ください

```
$ sudo apt install imagemagick
```

それでは、convert コマンドを使用して、画像ファイルの解像度とピクセルフォーマットを変換してみます。何らかの画像ファイルを用意してください。画像ファイルの解像度はフレームバッファの解像度とアスペクト比が同じである方が良いです (縦か横に伸びてしまうので)。

以下のコマンドで変換できます。

```
$ convert 元画像ファイル -resize 800x600! -depth 8 変換後ファイル名.bgra
```

フレームバッファ解像度 (800x600) 以外の画像ファイルは考えないためアスペクト比を無視して 800x600 へ変換するよう"-resize"オプションの解像度指定に"!"を付けています。また、BGRA(各色 8 ビット) へピクセルフォーマットを変換するため"-depth 8"オプションと変換後の画像ファイル名に".bgra"の拡張子を付けています。なお、ヘッダ等のメタデータが無い形式へ変換していますので、ファイルの中身は先頭からピクセルデータ (BGRA) が並びます。

6.3 1 枚の画像を表示してみる

まずは 1 枚の画像表示だけを行ってみます。

サンプルディレクトリは"061_iv_view_a_image"です。

前節で用意した画像ファイルがフレームバッファの解像度にもピクセルフォーマットにも既に合わせてあって、かつファイル先頭からピクセルデータが並ぶため、やることは単に以下の 2 点です。

1. 画像ファイルを開く
2. 画像ファイルのデータをフレームバッファへコピー

さっそく main.c を書き変えて行きたい所ですが、2. のコピー処理は「特定のメモリアドレスから特定のメモリアドレスへ指定されたバイト数をコピーする」という汎用関数として用意しておくと便利そうです。

そのため、先に memcpy という関数名で common.c と common.h へ関数を追加しておくことにします。

common.c の追加内容はリスト 6.4 の通りです。

▼**リスト** 6.4　061_iv_view_a_image/common.c

```
#include <common.h>

int strcmp(char *s1, char *s2)
{
        /* ・・・ 省略 ・・・ */
}

/* 追加（ここから） */
void memcpy(void *dst, void *src, unsigned long long size)
{
        unsigned char *d = (unsigned char *)dst;
        unsigned char *s = (unsigned char *)src;
        for (; size > 0; size--)
                *d++ = *s++;
}
/* 追加（ここまで） */
```

common.h の追加内容はリスト 6.5 の通りです。

▼**リスト** 6.5　061_iv_view_a_image/include/common.h

```
/* ・・・ 省略 ・・・ */

int strcmp(char *s1, char *s2);
void memcpy(void *dst, void *src, unsigned long long size);     /* 追加 */

#endif
```

これで必要な道具は用意できました。あとは main.c を変更するだけです (リスト 6.6)。

▼**リスト** 6.6　061_iv_view_a_image/main.c

```
/* ・・・ 省略 ・・・ */
#include <fs.h>
#include <common.h>     /* 追加 */

void start_kernel(void)
{
        /* ・・・ 省略 ・・・ */

        /* CPU の割り込み有効化 */
        enable_cpu_intr();

        /* 変更（ここから） */
        /* "image.bgra"を開いて表示 */
        struct file *img = open("image.bgra");
        memcpy(fb.base, img->data, img->size);
        /* 変更（ここまで） */

        /* halt して待つ */
        while (1)
                cpu_halt();
```

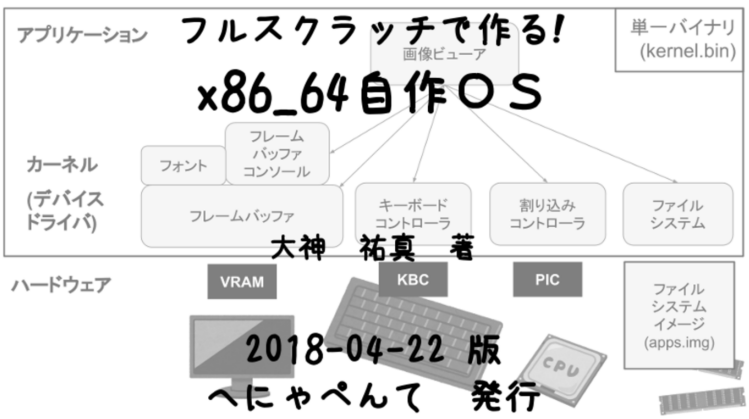
```
}
```

Makefile の変更点は無いです。make して kernel.bin を生成してください。

fs.img は、前節の convert コマンドで"image.bgra"という名前で画像ファイルを用意し、前章で作成した create_fs.sh を使用して生成します。

```
$ convert 元画像ファイル -resize 800x600! -depth 8 image.bgra
$ ./create_fs.sh image.bgra
$ ls fs.img
fs.img  # "fs.img"が生成される
```

実行すると図 6.4 のように画像が表示されます。ここでは表紙の元画像を表示してみています。



▲図 6.4　061_iv_view_a_image の実行結果

なお、main.c(リスト 6.6) にて memcpy 関数へ指定するコピーサイズ (第 3 引数) は struct file の size メンバを使用していました。そのため、例えばフレームバッファサイズが 640x480 な環境で実行する際は上記の convert コマンドと create_fs.sh コマンドで fs.img さえ作り直せば、kernel.bin は変更なしで画面表示が行えます。

6.4 ファイルシステムからファイルの一覧情報を取得する

これまでは open 関数を使用して「このファイル名のファイルをくれ」とファイルシステムへ問い合わせていました。しかし、画像ビューアを作成するにはファイルシステムに存在するファイルのリストを問い合わせる必要があります。まだこのような問い合わせにはファイルシステム (fs.c) が対応していないので、そのための機能追加を行います。

この節のサンプルディレクトリは"062_iv_ls"です。

やり方はいろいろとあるかと思うのですが、本書では以下のように fs.c へ実装することにします。

- unsigned long long get_files(struct file *files[])
- 引数: ファイルシステム内の各ファイルの struct file へのポインタを格納する領域を指定
- 戻り値: ファイルシステム内のファイルの個数

このような実装方針にした思想は脚注[*3]に記載します。これまでの実装も同じ思想に従っていたのですが、読者の方には関係ないので (お好きなように実装されれば良いので) 脚注へ記載します。

それでは、fs.c へ実装してみます。内容はリスト 6.7 の通りです。

▼**リスト** 6.7　062_iv_ls/fs.c

```
/* ・・・ 省略 ・・・ */

struct file *open(char *name)
{
        /* ・・・ 省略 ・・・ */
}

/* 追加（ここから） */
unsigned long long get_files(struct file *files[])
{
        struct file *f = fs_start;
        unsigned int num;

        for (num = 0; f->name[0] != END_OF_FS; num++) {
                files[num] = f;
                f = (struct file *)((unsigned long long)f + sizeof(struct file)
                                    + f->size);
        }
```

[*3] 自身が管理する機能の使い方を利用者が知らなくても良いように実装しています。例えば fs.c の場合、「各ファイルはアドレス上連続に並んでいること」や「ファイルシステムの最後は 0x00 で示すこと」は fs.c で隠ぺいすべきと考え、get_files() のような実装としています。

```
        return num;
}
/* 追加（ここまで） */
```

fs.h へプロトタイプ宣言の追加も行います (リスト 6.8)。

▼**リスト** 6.8　062_iv_ls/include/fs.h

```
/* ・・・ 省略 ・・・ */

void fs_init(void *_fs_start);
struct file *open(char *name);
unsigned long long get_files(struct file *files[]);     /* 追加 */

#endif
```

get_files 関数を使用して、試しに main.c でファイルシステム内のファイル名を表示してみます (リスト 6.9)。

▼**リスト** 6.9　062_iv_ls/main.c

```
/* ・・・ 省略 ・・・ */

void start_kernel(void *_t __attribute__ ((unused)), struct framebuffer *_fb)
{
        /* ・・・ 省略 ・・・ */

        /* CPU の割り込み有効化 */
        enable_cpu_intr();

        /* 変更（ここから） */
        /* ファイルシステム内のファイル名を表示 */
        struct file *files[10];
        int num = get_files(files);
        int i;
        for (i = 0; i < num; i++) {
                puts((char *)files[i]->name);
                puts("\r\n");
        }
        /* 変更（ここまで） */

        /* halt して待つ */
        while (1)
                cpu_halt();
}
```

試しに前章で"HELLO.TXT"と"FOO.TXT"を格納した fs.img で実行すると図 6.5 のように表示されます。

▲図 6.5　062_iv_ls の実行結果

6.5　画像ビューアを実装する

必要な道具はそろったので、画像ビューアを実装していきます。
この節のサンプルディレクトリは"063_iv_image_viewer"です。

6.5.1　実装方針

画像ビューアは iv.c と iv.h へ実装していくこととします。
そして、他の機能とやり取りするインタフェースとなる関数としては以下の 2 つを作成します。

- iv_init: 初期化処理。ファイルシステムからファイルリスト取得と、最初の 1 枚目の画像表示を行う
  - start_kernel 関数で各種初期化処理終了後に呼び出す
- iv_kbc_handler: キー入力に応じた処理を行う
  - KBC ハンドラから呼び出す

6.5.2　最初の画像表示処理を実装する (iv_init)

まず、iv_init 関数を実装します (リスト 6.10)。

▼リスト 6.10　063_iv_image_viewer/iv.c(iv_init())

```
#include <iv.h>
#include <fb.h>
#include <fs.h>
#include <common.h>
```

```
struct file *iv_files[MAX_IV_FILES];
unsigned long long iv_num_files;
unsigned long long iv_idx = 0;

void view(unsigned long long idx)
{
        memcpy(fb.base, iv_files[idx]->data, iv_files[idx]->size);
}

void iv_init(void)
{
        iv_num_files = get_files(iv_files);
        view(iv_idx);
}
```

iv_init 関数はファイルシステム内の全てのファイルを画像ファイルとして扱います ("ファイルシステム内には画像ファイルしか無いものとする"と仕様で決めた通りです)。また、iv.c の内部で使う関数としてファイルシステム上の idx 番目 (引数で指定) の画像表示を行う view 関数も作成しました。

iv.h はリスト 6.11 の通りです。

▼**リスト** 6.11　063_iv_image_viewer/include/iv.h

```
#ifndef _IV_H_
#define _IV_H_

#define MAX_IV_FILES   100

void iv_init(void);

#endif
```

リスト 6.11 で"MAX_IV_FILES"を 100 と定義しているため、画像ビューアで扱えるファイル数の上限は 100 です。100 という数に特に根拠はありませんが、増やす場合は poiboot.conf で設定した fs.img のロード先アドレスの領域が十分であるか UEFI Shell の memmap コマンドで確認してください[*4]。

そして、iv_init 関数を呼び出すよう main.c を書き変えます (リスト 6.12)。

▼**リスト** 6.12　063_iv_image_viewer/main.c

[*4] 800x600 の解像度の時、1 枚の BGRA 画像のサイズは 2MB 弱のため、100 枚の時はおおよそ 200MB 弱の領域が必要です。

```
/* ・・・ 省略 ・・・ */
#include <common.h>
#include <iv.h>         /* 追加 */

void start_kernel(void *_t __attribute__ ((unused)), struct framebuffer *_fb)
{
        /* フレームバッファ周りの初期化 */
        fb_init(_fb);
        set_fg(255, 255, 255);
        set_bg(0, 70, 250);
        /* clear_screen() 削除 */

        /* CPU 周りの初期化 */
        gdt_init();
        intr_init();

        /* 周辺 IC の初期化 */
        pic_init();
        kbc_init();

        /* 画像ビューアの初期化 */         /* 追加 */
        iv_init();                          /* 追加 */

        /* CPU の割り込み有効化 */
        enable_cpu_intr();

        /* 動作確認で書いていたコード削除 */

        /* halt して待つ */
        while (1)
                cpu_halt();
}
```

iv_init 関数で起動後最初の画像を表示するので、clear_screen 関数呼び出しは削除しました。

また、画像ビューア初期化処理中は割り込みは入ってほしくないため、割り込み有効化処理の直前で iv_init 関数呼び出しを行っています[*5]。

iv.c を追加しましたので、Makefile へも"iv.o"をビルド対象に追加しておきます (リスト 6.13)。

▼**リスト** 6.13　063_iv_image_viewer/Makefile

```
TARGET = kernel.bin
CFLAGS = -Wall -Wextra -nostdinc -nostdlib -fno-builtin -fno-common -Iinclude
LDFLAGS = -Map kernel.map -s -x -T kernel.ld
OBJS = main.o iv.o fbcon.o fb.o font.o kbc.o x86.o intr.o pic.o \  # iv.o を追加
```

[*5] これまで main.c へ書いていた各種の動作確認コードも厳密にはあまり割り込んで欲しく無い処理ではありますが、キーを押しっぱなしで起動とかでも無い限り特に問題は無いかと思います。(割り込まれたとしても文字が画面に出るか出ないかだけなので。)

```
        handler.o fs.o common.o

$(TARGET): $(OBJS)
# ・・・ 省略 ・・・
```

この時点で一度、動作確認してみると良いかと思います。("image.bgra"だけを格納した"fs.img"などを使うと良いです。)

6.5.3 KBC 割り込み時の処理を実装する (iv_kbc_handler)

それでは、割り込み時の処理を"iv_kbc_handler"という関数名で iv.c へ実装します (リスト 6.15)。

▼**リスト** 6.15　063_iv_image_viewer/iv.c(iv_kbc_handler())

```
/* ・・・ 省略 ・・・ */

struct file *iv_files[MAX_IV_FILES];
unsigned long long iv_num_files;
unsigned long long iv_idx = 0;  /* 追加 */

/* ・・・ 省略 ・・・ */

void iv_init(void)
{
        iv_num_files = get_files(iv_files);
        view(iv_idx);
}

/* 追加（ここから） */
void iv_kbc_handler(char c)
{
        switch (c) {
        case 'j':
                if (iv_idx < iv_num_files - 1)
                        view(++iv_idx);
                break;
        case 'k':
                if (iv_idx > 0)
                        view (--iv_idx);
        }
}
/* 追加（ここまで） */
```

リスト 6.15 にて、iv_kbc_handler 関数は押下されたキーが引数で渡される事を想定しています (後ほどそのように KBC のハンドラから呼び出します)。

iv_kbc_handler 関数内の処理としては、グローバル変数で用意した iv_idx(現在の画像のインデックス) を使用して、'j' キー押下で次の画像を表示し、'k' キー押下で前の画像を表示するようにしています。iv_idx が最後の画像を指していたり、最初の画像を指し

ていた場合は何もしないようにしています。

iv_kbc_handler 関数を KBC 割り込みハンドラから呼べるように iv.h へプロトタイプ宣言を追加します (リスト 6.15)。

▼**リスト** 6.15　063_iv_image_viewer/include/iv.h(iv_kbc_handler())

```
#ifndef _IV_H_
#define _IV_H_

#define MAX_IV_FILES   100

void iv_init(void);
void iv_kbc_handler(char c);    /* 追加 */

#endif
```

そして、KBC 割り込みハンドラ (do_kbc_interrupt 関数) で iv_kbc_handler 関数を呼び出すよう kbc.c を変更します (リスト 6.16)。

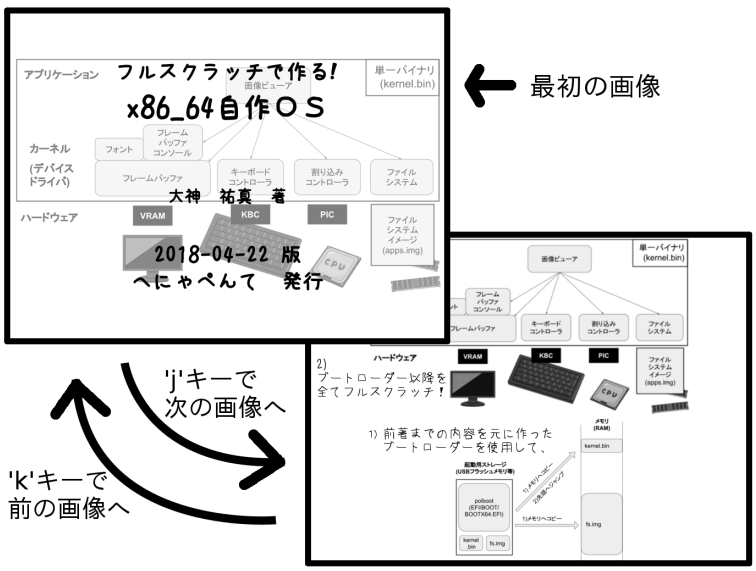
▼**リスト** 6.16　063_iv_image_viewer/kbc.c(iv_kbc_handler() **呼び出し**)

```
/* ・・・ 省略 ・・・ */
#include <pic.h>
#include <iv.h>         /* 追加 */

/* ・・・ 省略 ・・・ */

void do_kbc_interrupt(void)
{
        /* ステータスレジスタの OBF がセットされていなければ return */
        if (!(io_read(KBC_STATUS_ADDR) & KBC_STATUS_BIT_OBF))
                goto kbc_exit;

        /* make 状態でなければ return */
        unsigned char keycode = io_read(KBC_DATA_ADDR);
        if (keycode & KBC_DATA_BIT_IS_BRAKE)
                goto kbc_exit;

        /* KBC 割り込み処理を呼び出す */    /* 変更 */
        char c = keymap[keycode];
        iv_kbc_handler(c);              /* 変更 */

kbc_exit:
        /* PIC へ割り込み処理終了を通知 (EOI) */
        set_pic_eoi(KBC_INTR_NO);
}

void kbc_init(void)
{
        set_intr_desc(KBC_INTR_NO, kbc_handler);
        enable_pic_intr(KBC_INTR_NO);
}
```

これで追加・変更は終わりです。

convert コマンドで BGRA 画像を何枚か用意し、create_fs.sh スクリプトで fs.img を生成してください。

実行すると、画像が表示され、'j' キー/'k' キーで画像送り/戻りが確認できます (図 6.6)。ここでは表紙・裏表紙の元画像を表示してみています。



▲図 6.6　画像ビューア実行の流れ

割り込みハンドラからアプリケーションの機能を呼び出す構成について

趣味で書く OS は何をどうやるのも自由なので、設計についても自分なりに考えてみる事は面白いです。

今回はやりたいことを実現する手っ取り早い方法として kbc.c(KBC ドライバ) 内の KBC 割り込みのハンドラから画像ビューアの割り込み処理を呼び出すようにしましたが、設計としては微妙な気がします。

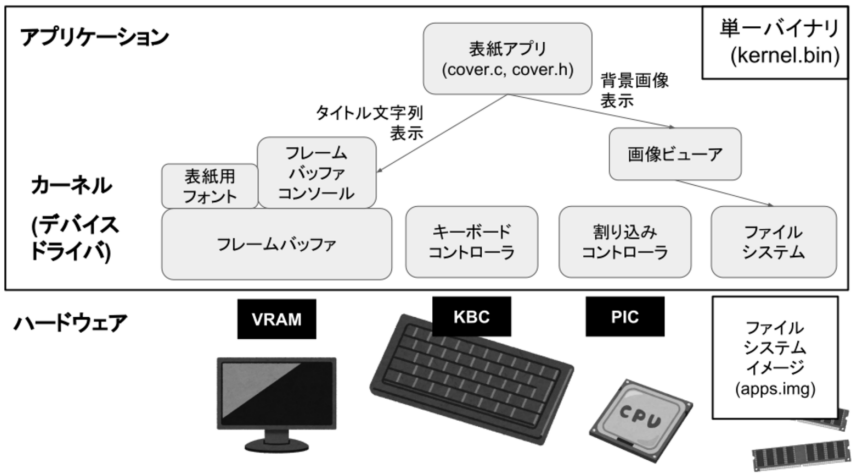
KBC ドライバは KBC を管理・制御する事に従事すべきで、「画像ビューア」等という上位のアプリケーション (カーネルと一緒になってしまってはいますが) と直接やり取りするべきでは無いかと思います。ドライバが直接やり取りする対象はカーネルなので、修正するとすれば、例えば、「割り込み時のコールバック関数を登録する機能をカーネル側に用意し、アプリケーションはカーネルへ『この関数を割り込み時に呼び出してくれ』とお願いする」等でしょうか。

もっと考えると、ハードウェアの機能である「割り込み」をアプリケーションに扱わせるのも微妙かもしれません。アプリケーションは「割り込みを使いたい」のではなく単に「データが欲しい」だけな場合があります。今回の場合、「キー入力があった事とその入力値」を知ることができれば割り込みで無くとも何でも構わないのです。そのため、C 言語でアプリケーションを作るときの getc 関数のように「アプリケーションへは当該データを得るためのインタフェースを提供し、割り込み等の処理はカーネル内へ隠蔽する」というのも良いかと思います。

付録 A

表紙について

本書の表紙は 6 章最後のサンプル (063_iv_image_viewer) を改造して作っています (サンプルディレクトリ"A00_cover_app")。構成は図 1.1 の通りです。



▲図 A.1　表紙アプリの構成

表紙アプリは、画像ビューアの機能を使って画像を表示し、表紙用フォントを使ってタイトルの文字列を描画しています。表紙アプリのコード自体は特に目新しいことをしているわけではないので特に紹介はしません。

A.1　フォントジェネレータ (create_font.sh)

表紙アプリを作るためにフォントを追加する際、文字が書かれた画像から font.c と font.h を生成するシェルスクリプトを作成しました。"create_font.sh"というファイル名で"A00_cover_app"ディレクトリ内に置いてあるので、興味があれば使ってみてください。

■ 使い方

まず、1 文字毎に 1 枚の画像を用意し、専用のディレクトリへ格納してください。ここでは"src"というディレクトリへ格納したとします。

あとは、create_font.sh を実行するだけです。

実行例

```
$ ./create_font.sh src dst 24 32
```

上記のように実行した場合、src ディレクトリ内の画像について生成した font.c と font.h が dst ディレクトリへ格納されます。第 3・4 引数はフォントの幅と高さで、ここでは幅 24px、高さ 32px を指定しています。

font.c の font_bitmap 配列は、各文字のインデックスが"FONT_<元画像ファイル名>"という定数となるように生成されます。"FONT_<元画像ファイル名>"の定数は font.h に enum で宣言されていますので、font_bitmap 配列内の文字の並びを調整する場合は font.h の enum 宣言部分を変更してください。どう変更すれば良いかについては"A00_cover_app"内の font.h を見てみてください。

付録 B

poiboot について

本書で使用している"poiboot"は前著までの「フルスクラッチで作る!UEFI ベアメタルプログラミング」シリーズの知見 + α で作成した簡単なブートローダーです。

ここでは poiboot の構成と、前著からの + α の部分、加えてブートローダー作成に際して筆者が遭遇した問題について説明します。

B.1 poiboot の構成

poiboot はサンプルディレクトリ内の"A01_poiboot"ディレクトリに格納しています。

ソースディレクトリの構成はリスト 2.1 の通りです。

▼**リスト** 2.1　poiboot **のディレクトリ構成**

```
A01_poiboot/
├── Makefile
├── poiboot.c
├── libuefi/
└── include/
```

前著までで紹介した UEFI ファームウェアを直接扱うためのコードは"libuefi"という名前でライブラリ化し、ライブラリ本体のソースコードを"libuefi"ディレクトリに、ヘッダを"include"ディレクトリに配置しています。そして、libuefi を使用してブートローダーを作っている poiboot の本体が poiboot.c です。

poiboot.c では以下の処理を行っています。

1. libuefi 初期化
2. 設定ファイル (poiboot.conf) 読み込み
3. kernel.bin を RAM へロード

4. fs.img が存在すれば RAM へロード
5. カーネルへ渡す情報の準備
6. UEFI の boot services を終了させる
7. カーネルへジャンプ

「1. libuefi 初期化」は、前著で説明した efi_init 関数です。

「2. 設定ファイル (poiboot.conf) 読み込み」では設定ファイル"poiboot.conf"をパースし、カーネル先頭アドレス (kernel.bin を配置するアドレス) とファイルシステム先頭アドレス (apps.img を配置するアドレス) を設定します

「3. kernel.bin を RAM へロード」では、2.2 節で説明したカーネルのバイナリフォーマットを解釈し、RAM へロードします。といっても、2.2 節で説明した通り、やることは、1) カーネル本体を poiboot.conf 設定 (本書では 0x00110000 を想定) のアドレスへロードし、2) ヘッダの情報に従い bss セクションをゼロクリアの 2 つです。ファイルを扱う過程で UEFI ファームウェアのファイル操作を使用しますが、UEFI でのファイル操作は前著までで説明した通りです。

「4. fs.img が存在すれば RAM へロード」は、"fs.img"という名前のバイナリが存在すれば、poiboot.conf 設定 (本書では 0x0000000100000000 を想定) のアドレスへロードします。

「5. カーネルへ渡す情報の準備」について、ブートローダーからカーネルへ渡す情報は SystemTable・fb・fs の先頭アドレスです。SystemTable は efi_main 関数の第 2 引数で得られるものです。fb はカーネルへフレームバッファの情報を渡すために用意した struct fb 構造体の変数で、init_fb 関数で作成しています。fs は poiboot.conf の設定値が渡ることになります。

「6. UEFI の boot services を終了させる」では、BootServices の ExitBootServices 関数を呼び出し、UEFI ファームウェアが様々なリソースを管理しているモードを終了します。ExitBootServices を呼び出す処理は前著まででは説明していない + α の内容で、次節で説明します。

「7. カーネルへジャンプ」では、SystemTable と struct fb とファイルシステムの先頭アドレスを関数の引数として受け渡すためにそれぞれをレジスタへ格納します。関数呼び出しの第 1・第 2・第 3 引数はそれぞれ、RDI レジスタ・RSI レジスタ・RDX レジスタで受け渡しますので、それぞれのレジスタに SystemTable と fb と fs の先頭アドレスを格納します。そして、スタックポインタを設定し、カーネルへジャンプします。

B.2 前著の + α (ExitBootServices)

BootServices の ExitBootServices 関数の呼び出し方は少々特殊です。といっても仕様書に書いてあるとおりで、複雑なものではないです。

ExitBootServices 関数を呼び出すために必要な手続きと ExitBootServices 関数の呼び出しは exit_boot_services 関数にまとめています。exit_boot_services 関数の実装はリスト 2.2 の通りです。

▼**リスト** 2.2　A01_poiboot/libuefi/mem.c

```
void exit_boot_services(void *IH)
{
        unsigned long long status;
        unsigned long long mmap_size;
        unsigned int desc_ver;

        do {
                mmap_size = MEM_DESC_SIZE;
                status = ST->BootServices->GetMemoryMap(
                        &mmap_size, (struct EFI_MEMORY_DESCRIPTOR *)mem_desc,
                        &map_key, &mem_desc_unit_size, &desc_ver);
        } while (!check_warn_error(status, L"GetMemoryMap"));

        status = ST->BootServices->ExitBootServices(IH, map_key);
        assert(status, L"ExitBootServices");
}
```

ExitBootServices 関数は GetMemoryMap 関数で取得したメモリマップのキー (map_key) を取得した後、他の UEFI の機能を呼び出したりすること無く ExitBootServices 関数を呼び出す必要があります。そのため、GetMemoryMap 関数と ExitBootServices 関数の呼び出しをひとつの関数にまとめています。

なお、ExitBootServices を呼び出すと、それ以降、BootServices 内の関数群や各種プロトコル[*1]は使用不可となりますが、RuntimeServices 内の関数群は引き続き使用可能です。

ブートローダーは UEFI ファームウェアから物理アドレスのメモリマップを取得できるので、カーネルが動作し始める時のスタックポインタ設定もブートローダーの役割としています。

[*1] 実体は、関数ポインタをメンバに持つ構造体

B.3 UEFI のファイル読み出しが正しく動作しないケースがありました

ブートローダーが UEFI の機能を利用してファイル読み出しを行う際は、"EFI_FILE_PROTOCOL"というプロトコルが持つ"Read"関数を使用します。

Read 関数には、引数で「読み出して欲しいバイト数」を変数のポインタとして渡し、Read 関数は「実際に読み出したバイト数」を渡されたポインタが指す変数へ格納して返します。

問題となったのが、Read 関数が返す「実際に読み出したバイト数」です。Lenovo 製 ThinkPad E450 にて、640x480 の BGRA 画像 2 枚を格納した fs.img(2.4MB 程) を Read させたとき、「実際に読み出したバイト数」としては指定した fs.img のサイズを返すのですが、読み出し先を見てみると途中からでたらめなデータが格納されていることがありました。

100% 読み出しに失敗するわけではなく、成功する場合もあり、因果は不明ですが筆者が試す限り 100% 問題が再現する手順は以下の手順でした。

1. PC を立ち上げ、GRUB メニューが表示されるまで待つ
2. USB フラッシュメモリ挿入
3. GRUB メニューで'c' キーを押下し GRUB のシェルを立ち上げる
4. GRUB シェル内で"reboot"コマンドにより再起動
5. USB フラッシュメモリから起動した poiboot で Read に失敗

USB フラッシュメモリの挿入タイミングはいつでも良いです。UEFI ファームウェアの機能を利用するプログラム (本書の場合 poiboot) を「PC の再起動で立ち上げる」事がきっかけのようで、Debian が起動している状態で USB フラッシュメモリを挿入し、再起動したときに発生することもありました。しかし、一度シャットダウンした PC に USB フラッシュメモリを挿入し PC の電源を入れた場合に発生したことは無かったように思います。

上記の再現手順においても kernel.bin(16KB 程度) の Read はちゃんと最後まで読み出せていたので、16KB ずつ Read するようにワークアラウンドを作成しました。それが libuefi 内 file.c の safety_file_read 関数です (Read する単位は file.c に"SAFETY_READ_UNIT"という定数で設定)。

筆者の環境の場合、16KB までは 1 回の Read で正しく読み出せていたので「16KB ずつ」としていますが、その他の UEFI ファームウェアではより小さいサイズしか正しく読み出せない (なおかつ、読み出したサイズは指定された通りのサイズを返す) 場合もある

かもしれません。16KB は"063_iv_image_viewer"の kernel.bin のサイズでもあるので、その場合は kernel.bin 自体正しく読み出せていない可能性があります。そのため、「挙動がおかしいが kernel.bin や fs.img 自体には問題が無いと思われる」場合は、poiboot で kernel.bin と fs.img を正しくメモリへロードできているか確認してみてください。

poiboot が画面へ出すログを見る場合は、main.c で無限ループのみを行う kernel.bin を用意すれば良いです。poiboot はカーネル起動時に画面のクリアをしないため、カーネル側でフレームバッファをクリアしない限りは poiboot が画面へ出力したログを見ることができます。

poiboot が画面へ出力するログの例は図 2.1 の通りです。

```
Boot Failed. EFI DVD/CDROM
Boot Failed. EFI Floppy
Boot Failed. EFI Floppy 1
Starting OS5 UEFI bootloader ...
conf_size: 0x0000000000000022
load conf ... done
kernel_start: 0x0000000000110000
stack_base: 0x0000000000210000
apps_start: 0x0000000100000000
kernel_size: 0x0000000000004118
load kernel head ... done
load kernel body ... done
kernel body first 16 bytes: 0x55 48 89 E5 48 83 EC 20 48 89 7D F8 48 89 75 F0
kernel body last 16 bytes: 0x00 41 0E 10 86 02 43 0D 06 02 48 0C 07 08 00 00
root->Open(apps):800000000000000E
apps load failure. skip.
kernel_arg1: 0x00000000BFA43F18
kernel_arg2: 0x0000000000407180
kernel_arg3: 0x0000000000000000
```

▲図 B.1　poiboot が画面へ出力するログの例

"kernel body"と"apps"のそれぞれでロードした最初の 16 バイト ("first 16 bytes") と最後の 16 バイト ("last 16 bytes") を出力しています。それらが実際の kernel.bin と fs.img と合っているか確認してください。なお、"kernel body"は kernel.bin のヘッダ (BSS の先頭アドレスとサイズ) を除いた部分です。kernel.bin の先頭 32 バイト (ヘッダ) を除いた以降 (33 バイト目以降) が"kernel body first 16 bytes"に並びますので注意してください。

なお、上記に「Lenovo 製 ThinkPad E450 にて」と書いている通り、QEMU ではこの問題は発生しませんでした。E450 以外の実機は筆者の所持している PC の都合上確認できていません。

付録 C

GPT フォーマット方法

UEFI ファームウェアは、起動用のストレージ等の GPT あるいは FAT フォーマットされた第 1 パーティションを認識します。そのため、GPT あるいは FAT でフォーマットしていないと UEFI ファームウェアが"EFI/BOOT/BOOTX64.EFI"という名前で配置したブートローダー (poiboot) を見つけてくれません。

ここでは旧来の MS-DOS パーティションテーブルではなく、UEFI で新仕様として加わった GPT(GUID Partition Table) 形式で USB フラッシュメモリをフォーマットする方法を紹介します。

GPT 形式のフォーマットに、ここでは gdisk コマンド (fdisk の GPT 対応版) と mkfs を使用します。

まずは gdisk で GPT のパーティションテーブルを作成します (USB フラッシュメモリ第 1 パーティションが"/dev/sdb1"として認識されているとします)。

```
$ sudo gdisk /dev/sdb
GPT fdisk (gdisk) version 0.8.10

Partition table scan:
  MBR: protective
  BSD: not present
  APM: not present
  GPT: present

Found valid GPT with protective MBR; using GPT.

Command (? for help): o  ← GPT を作成する指定
This option deletes all partitions and creates a new protective MBR.
Proceed? (Y/N): Y

Command (? for help): n
Partition number (1-128, default 1):
First sector (34-31277022, default = 2048) or {+-}size{KMGTP}:
```

```
Last sector (2048-31277022, default = 31277022) or {+-}size{KMGTP}:
Current type is 'Linux filesystem'
Hex code or GUID (L to show codes, Enter = 8300): ef00
Changed type of partition to 'EFI System'

Command (? for help): p
Disk /dev/sdb: 31277056 sectors, 14.9 GiB
Logical sector size: 512 bytes
Disk identifier (GUID): 2DDA5A7F-F45A-4B9C-9AB1-137FC2856D83
Partition table holds up to 128 entries
First usable sector is 34, last usable sector is 31277022
Partitions will be aligned on 2048-sector boundaries
Total free space is 2014 sectors (1007.0 KiB)

Number  Start (sector)    End (sector)  Size       Code  Name
   1            2048        31277022   14.9 GiB    EF00  EFI System

Command (? for help): w

Final checks complete. About to write GPT data. THIS WILL OVERWRITE EXISTING
PARTITIONS!!

Do you want to proceed? (Y/N): Y
OK; writing new GUID partition table (GPT) to /dev/sdb.
The operation has completed successfully.
```

"o"コマンドで GPT を作成する指定をし、USB フラッシュメモリの全容量を使用する単一のパーティションを作成しています。"ef00"は"EFI System Partition"という UEFI 起動ディスク専用のパーティションタイプです。

次に、作成したパーティションを FAT32 でフォーマットします。

```
$ sudo mkfs.vfat -F 32 /dev/sdb1
mkfs.fat 3.0.27 (2014-11-12)
```

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 OS(特にカーネル) 自作の第一歩として、画面出力やキー入力といった入出力のデバドラ、割り込み、簡易的なファイルシステムといったものをカーネルとして抽象化する方法をステップバイステップで紹介しました。本書で抽象化を行ったこれらの機能を組み合わせるだけでも「ちょっとしたゲーム」や、「自作 OS でプレゼン[*1]」なんてこともできるかと思います。

ただし、本書で扱ったデバイスは 64 ビット以前から存在するとてもレガシーなものです。キーボード入力一つ考えても今や USB が主流であり、今後はこのような今主流のデバイスを自作 OS でも使えるようにするのも面白い (なおかつ他の人にも役立つ) のかもしれません。

何にしても、趣味ならば楽しい方が良いので、自作 OS であれベアメタルプログラミングであれ、楽しいと思うやり方でやっていけたら良いのかなと思います。本書が自作 OS(というよりベアメタルプログラミング?) を楽しむ一つの助けになれば幸いです。

[*1] ノート PC の BIOS(UEFI) 設定でプライマリのディスプレイを内蔵ディスプレイから外部ディスプレイへ切り替えれば良いです。

参考情報

参考にさせてもらった情報

- Intel 64 and IA-32 Architectures Software Developer Manuals
  - `https://software.intel.com/en-us/articles/intel-sdm`
  - x86 CPU を扱う上での一次情報です
- 書籍「パソコンのレガシィ I/O 活用大全」(CQ 出版)
  - KBC や PIC 等、PC/AT 互換機で標準的 (ただし今や古い) な IC の使い方についてはほとんどこの本を参考にさせていただきました

本シリーズの過去の著作

- フルスクラッチで作る!UEFI ベアメタルプログラミング! パート 1、パート 2
  - パート 1/2 共に以下の URL から PDF 版は無料でダウンロードできます
  - `http://yuma.ohgami.jp`

フルスクラッチで作る!x86__64 自作 OS

2018 年 4 月 22 日　技術書典 4 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