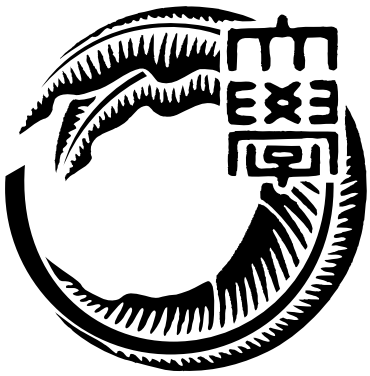
令和2年度 卒業論文

Gears OS の Bootに関する研究



琉球大学工学部工学科知能情報コース

175701G 氏名 奥田　光希

指導教員：河野 真治

目 次

図 目 次

表 目 次

ソースコード目次

第1章　初めに

1.1　要旨

2017年にIntel社が2020年までにLegacy BIOSとUEFIへの互換を非推奨とし、互換モジュールのCSMを削除すると発表した。[1] Legacy BIOSは長年に渡り16bitパソコンの時代からの資産を引き継いできたため、16bitモードでしか動作しない。そのためPCの進化に伴い、致命的な問題点が発生する。問題点として、拡張性がないことがあげられる。EthernetやUSBにつながるでディスクなど、新たにブートデバイスが追加されるたびに, OSのブートローダを変更しなければならない。またマザーボードごとに、ファームウェアをアセンブラで開発する必要がある。また、1MBのメモリ制限により、セキュリティを含めたシステム機能の強化が困難であるためセキュリティにも問題がある。これらの問題を解決するためにUEFIが開発された。UEFIは、2TBを超える大きなディスクからブートでき、高速にブートできる。CPUに依存しないアーキテクチャとドライバを持ちネットワークも使用可能な柔軟なプレOS環境が利用できる。今後、Legacy BIOSからUEFIへの移行が急速に進むだろう。

当研究室では、信頼性と拡張性をテーマにGearsOSを開発している。GearsOSはContinuation based C(CbC)によってアプリケーションとOSそのものを記述している。現在、CbCで証明可能なOSを実装するために、xv6のCbCの書き換えを行っている。xv6はLegacy OSなため、UEFIから起動することができない。UEFIからxv6を起動させることができれば、拡張性が大きく広がる。本研究では、ARMで動くシングルコンピュータであるRaspberryPi上にUEFIからGearsOSをブートさせることを目指している。

1.2　論文の構成

第2章　Continuation based C(CbC)

2.1　Continuation based C(CbC)

Continuation based C(CbC)[2] は、当研究室で開発されているプログラミング言語である。CbC は、C 言語の下位言語であり、関数呼び出しではなく継続を導入している。CbC では、関数の代わりに CodeGear という単位でプログラミングを行う。CodeGear は入力と出力を持ち、CbC では引数が入出力になっている。図 2.1 の様に CodeGear から次の CodeGear へと goto による継続で遷移して処理を行い、引数として出力を与える。

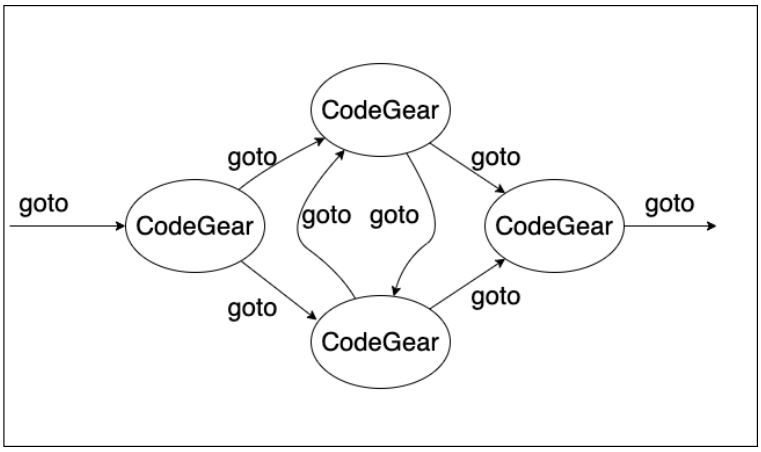


図 2.1: CbC の遷移

CbC には、GCC[3] 上に実装されたものと LLVM/Clang[4] 上に実装されたものがある。GCC と LLVM/Clang で実装された CbC を ARM で Compile するには Cross Compile を行う必要がある。GCC と LLVM では CbC を動かすのに GCC の方が安定しているので CbC の GCC の CrossCompile 環境を作成する。この Cross Compile 環境は Singularity で作成した。

2.1.1　GCC

GCC は GNU Compiler Collection の略で GNU プロジェクトが開発及び配布している、C/C++/Object-C などのプログラミング言語のコンパイラ集のことである。

2.1.2 LLVM/Clang

LLVM とは、モジュラー構成及び再利用可能なコンパイラとツールチェーン技術などを開発するプロジェクトの名称である。Clang は LLVM をバックエンドとして利用する C/C++/Object-C のコンパイラである。

2.1.3 CrossCompile

Cross Compile は、Compiler が動作している以外のプラットフォーム向けに実行ファイルを生成する機能を持った Compile 手法である。Raspbery Pi でいうと、Raspbian 以外の OS 環境であらかじめ Raspberry Pi で CbC が動くように Cross Compile を行う。そこで生成されたコードを Raspberry Pi に移すことで、実行できる様になる。

2.1.4 Singularity

Singularity[5] とは、ユーザーが自身の計算環境を完全再現し、保持できる様にした Linux コンテナである。Singularity はマルチユーザーに対応していて、コンテナ内の権限は実行ユーザーの権限を引き継ぐ。そのため、ユーザーに特別な権限の設定が必要ない。また、複雑なアーキテクチャとワークフローをサポートできるよう設計されていて、ほぼ全ての環境に適応できる。さらに、デフォルトで、HOME,/tmp,/proc,/sys,/dev がコンテナにマウントされ、サーバー上の GPU を簡単に利用できる。コンテナイメージは Singularity Image Format(sif) と呼ばれる単一ファイルベースなため、アーカイブや共有が容易である。

2.2 CbC on GCC CrossCompile

Singularity で環境を作成するためにファイル 2.1 を作成する。

ファイル 2.1: CbC_gcc_cross.def

```
1  BootStrap: docker
2  From: ubuntu:20.04
3
4  %post
5      apt-get update
6      apt-get upgrade -y
7      DEBIAN_FRONTEND=noninteractive \
8      apt-get install -y \
9          mercurial \
10         build-essential \
11         wget \
12         libmpc-dev \
13         libgmp-dev \
14         libmpfr-dev \
15         flex \
16         libssl-dev \
17         ninja-build \
18         g++-multilib \
19         gcc-multilib \
```

```
20          cmake \
21          ninja-build
22      DEBIAN_FRONTEND=noninteractive \
23      apt-get install -y \
24          crossbuild-essential-armhf
25
26      # install cbc_gcc
27      hg clone http://www.cr.ie.u-ryukyu.ac.jp/hg/CbC/CbC_gcc/
28      mkdir -p /usr/local/cbc_gcc
29      mkdir -p /CbC_gcc/builddir
30      cd /CbC_gcc/builddir
31      sh /CbC_gcc/configure -v --prefix=/usr/local/cbc_gcc --target=arm-linux-
            gnueabihf \
32          --enable-languages=c,lto --with-arch=armv7-a --with-fpu=vfpv3-d16 --with-
                float=hard \
33          --disable-multilib --disable-nls --enable-checking=tree,rtl,assert,types "
                CFLAGS=-g3␣-O0" \
34          --disable-libstdcxx --disable-libssp --disable-libstdcxx-pch --disable-
                libmudflap \
35          --with-newlib --with-sysroot=/ --with-as=/usr/arm-linux-gnueabihf/bin/as
                --with-ld=/usr/arm-linux-gnueabihf/bin/ld
36      ./config.status
37      make -j$(nproc) all-gcc
38      make install-gcc
39
40      # setup linker
41      ln -s /usr/arm-linux-gnueabihf/lib/crt*.o /usr/local/cbc_gcc/lib/gcc/arm-
            linux-gnueabihf/10.0.1/ && \
42      ln -s /usr/lib/gcc-cross/arm-linux-gnueabihf/9/crt*.o /usr/local/cbc_gcc/lib/
            gcc/arm-linux-gnueabihf/10.0.1/ && \
43      ln -s /usr/lib/gcc-cross/arm-linux-gnueabihf/9/libgcc* /usr/local/cbc_gcc/lib
            /gcc/arm-linux-gnueabihf/10.0.1/
44
45      # download arm-none-eabi
46      wget "https://developer.arm.com/-/media/Files/downloads/gnu-rm/10-2020q4/gcc-
            arm-none-eabi-10-2020-q4-major-x86_64-linux.tar.bz2?revision=ca0cbf9c-9
            de2-491c-ac48-898b5bbc0443&la=en&hash=68760
            A8AE66026BCF99F05AC017A6A50C6FD832A" -O /tmp/arm.tar.bz2 && \
47      tar -jxvf /tmp/arm.tar.bz2 -C /opt && \
48      mv /opt/gcc-arm-none-eabi-10-2020-q4-major /opt/tools
49
50  %environment
51      export LANG=C
52      export PATH=/usr/local/cbc_gcc/bin:$PATH
```

def ファイルが作成できたら singularity build を下記のように行う。

コマンド 2.2: singularity build

```
1  singularity build --fakeroot cbc_gcc_cross.sif cbc_gcc_cross.def
```

build で生成された sif ファイル 2.3 で CrossCompile を行う。例としてソースコード 2.4 を動かす。

ファイル 2.3: singularity 上で CrossCompile

```
1  singularity shell cbc_gcc_cross.sif
2  Singularity> arm-linux-gnueabihf-gcc src/hello.cbc
3  Singularity> file a.out
4  a.out: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically
       linked, interpreter /lib/ld-linux-armhf.so.3, for GNU/Linux 3.2.0, not
       stripped
```

CrossCompile により生成された a.out は Raspberry Pi で実行することができる。

ソースコード 2.4: hello.cbc

```
1  #include<stdio.h>
2
3  __code hello(){
4          print("hello,␣world!\n");
5  }
6
7  int main(void){
8          goto hello();
9          return 0;
10 }
```

第3章　Raspberry Pi上のGearsOS

3.1　GearsOS

GearsOS[6] は当研究室で信頼性と拡張性をテーマに開発されているOSである。GearsOSはContinuation based C(CbC) によって記述されている。当研究室では、GearsOSの信頼性とCbCの有効性を示すために、基本的な機能を揃えたOSであるxv6をCbCで置き換えを行っている。これにより、OSのここのシステムコールを持つ状態を明確にすることができると考えている。CbCで書き換えられたxv6をRaspberry Piに搭載することでハードウェア上でのメタレベルの計算や並列実行を行える様になる。

3.2　xv6

xv6[7] とは、マサチューセッツ工科大の大学院生向け講義の教材として使うために、UNIX V6 という OS を ANSI-C(規格化された C 言語) に書き換え、x86 に移植した xv6 OS である。x86 アーキテクチャで動作する。xv6 はプロセス、仮想メモリ、カーネルとユーザの分離、割り込み、ファイルシステムなどの基本的な Unix の構造を持つにも関わらず、シンプルで学習しやすい。

3.3　Raspberry Pi

Raspberry Pi[8] は、ARM プロセッサを搭載したシングルコンピュータ。Raspberry Piにはいくつか種類があり、本研究では Raspberry Pi 3 Model B を仕様する。Raspberry Pi 3 Model B には、USB2.0 コネクタが 4 つ、microSD カードスロット、HDMI 出力、40 ピン GPIO などがついている。CPU は、ARM アーキテクチャの Cortex-A53 で CPU クロックは 1.4GHz でメモリは 1 GB ある。

3.4　Raspberry Pi 上の xv6

xv6 は x86 で動作する OS である。Raspberry Pi 上で xv6 を動かすためには、ARM に対応した xv6 を用意する必要がある。そのため Raspberry Pi 用に移植した xv6-rpi を用いる。Raspberry Pi 上で起動している xv6 に入力を行うために USB シリアルケーブルで MacBook と接続する。その時、図 3.1 の様に Raspberry Pi の 6 番ピン (黒)、8 番ピン (白)、

10 番ピン (緑) の 3 つを USB シリアルケーブルで接続する。この時、HDMI でディスプレイに接続しておく。Mac 側では、USB シリアルケーブルのドライバをインストールして Raspberry Pi と接続すると、dev ディレクトリ直下に tty.usbserial として認識される。Mac 側でコマンド 3.1 を使い、シリアル通信を行うと、Mac のキーボードから入力を行える様になる。この時、コマンド 3.1 を打ってから、Raspberry Pi に電源を入れないと正常に起動しない。

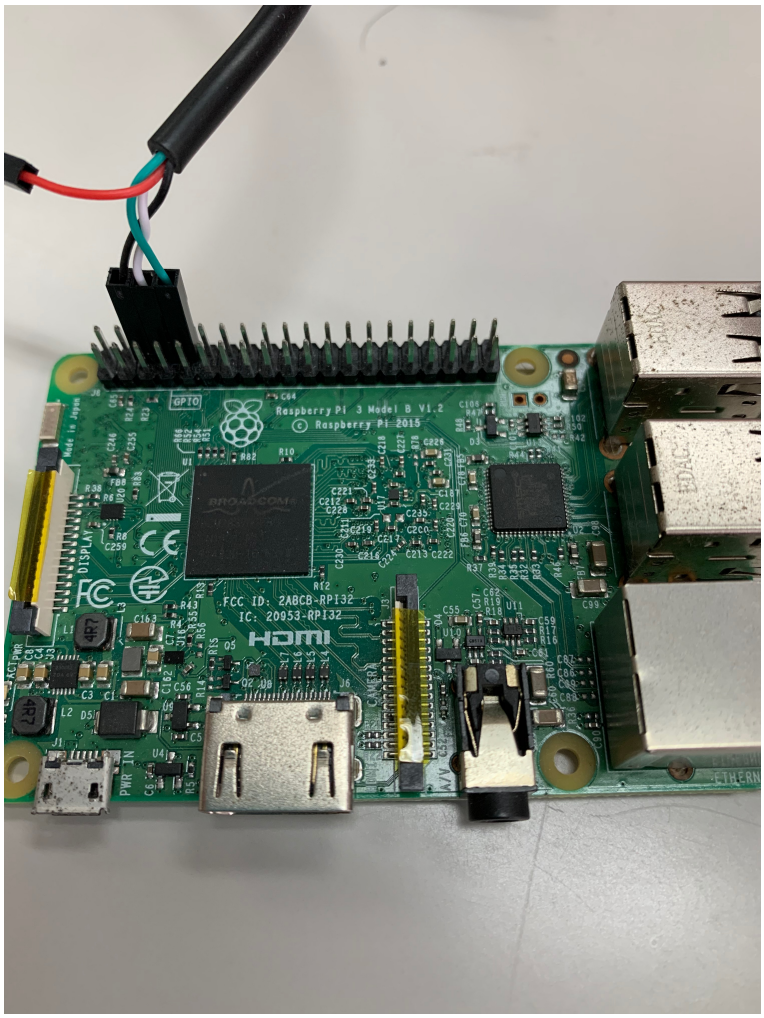


図 3.1: Raspberry Pi と USBserial 接続

コマンド 3.1: screen コマンド

```
1 screen /dev/tty.usbserial-143130 1152000
```

第4章　UEFI

4.1　UEFI

UEFI[9] とは、Unified Extensible Firmware Interface の略で OS とプラットフォームファームウェアの間のソフトウェアインタフェースを定義する仕様である。1990 年代半ばに Intel によって BIOS に変わる EFI 仕様が開発された。2005 年に Intel、AMD、Apple、Microsoft などの企業からなる Unified EFI　Forum という業界団体のもと UEFI が開発された。UEFI は単なるインタフェースの仕様であるため、特定のプロセッサに依存しない。以前までの BIOS と異なり、近代的なソフトウェア開発手法を用いることが推奨されていて、C 言語などで実装ができる。

4.2　BIOS と UEFI

UEFI は BIOS の後継として開発されたが BIOS と大きな違いがいくつかある。BIOS と UEFI の違いは表 4.1 である。

表 4.1: BIOS と UEFI の違い

	BIOS	UEFI(32bit)	UEFI(64bit)
CPU	16bit	32bit	64bit
メモリ空間	1MB	4GB	256TB
BOOT	MBR	GPT	GPT
デバイス規格	PS/2	USB	USB

4.2.1　CPU

BIOS は、40 年近く前から存在しているので、16bitCPU 前提のアーキテクチャであるため 16bit で起動する。また、CPU リアルモードでないと BIOS から起動できない。さらに、CPU のアーキテクチャに依存し、アーキテクチャごとに設定しなければならない。しかし、UEFI は、32bit、64bit の両方を起動できる。起動も 64bit モードで可能。また、CPU のアーキテクチャに依存しない。

4.2.2 メモリ空間

16bitCPU のメモリのアドレス空間は 2 の 16 乗で 64KB であった。つまり、16bitBIOS では、64KB の 16 倍である 1MB までしか使えない。UEFI では、32bit なら 2 の 32 乗 bit で 4GB、64bit なら 2 の 64 乗 bit で 256TB までメモリを潤沢に使える。これにより、セキュリティを含めたシステム機能の強化が可能になった。

4.2.3 BOOT

BIOS と UEFI では、BOOT 方式が違う。BIOS は、ディスク先頭の 512 バイトに BootLoader とパーティーションテーブル (MBR) が格納されていて第一セクタの 512 バイトがメモリにコピーされ、そこにジャンプする。そして、その BootLoader が起動する。BootLoader がカーネルとディスクイメージをメモリにロードし、カーネルが初期化処理をする。その後 OS が起動される。UEFI は、UEFI ファームウェアがロードされ、起動に必要なハードウェアを初期化する。次にファームウェアが Boot マネージャのデータを読み込みどの UEFI Application をどこから起動するか決定する。ファームウェアのブートマネージャのブートエントリに定義されているように UEFI　Application をファームウェアが起動する。起動しら UEFI Application が他の Application やカーネルと BootLoader を起動する。

4.2.4 デバイス

マウスやキーボードなどのデバイスの規格が BIOS と UEFI で変わる。BIOS は USB が発明される前から存在しているのでデバイス規格は PS/2 を使用していた。PS/2 は端子を通じてキーボードとマウスがキーボードコントローラに接続され、CPU から I/O バス経由でキーボードコントローラとやりとりをする。しかし、PS/2 ではマウスとキーボードしか接続できないため、汎用性の高い USB が主流になってきた。だが、USB に対応していないデバイスも存在したため、USB キーボードを PS/2 キーボードに見せかけるエミュレータ機能が存在した。一方、UEFI では USB が主流なため、デバイスの規格は基本的に USB であることが多い。

4.3 UEFI　Application

aaa

4.4 UEFI Hello World

UEFI を開発する際に

4.5 Raspberry Pi 上の UEFI

a

4.6 QEMU 上の UEFI

第5章　Boot Loader

5.1　bootloader

aaa

第6章　今後の課題

6.1　今後の課題

aaaa

参考文献

[1] Intel/Unified EFI Forum, https://www.uefi.org ,2017/11/3.

[2] 宮城光希, 河野慎治.CbC 言語による OS 記述. 琉球大学工学部情報工学科平成 29 年度学位論文 (学士),2017.

[3] GNU Compiler Collection (GCC) Internals,https://gcc.gnu.org/onlinedocs/gccint/

[4] Clang: a C language family frontend for LLVM,https://clang.llvm.org.

[5] https://sylabs.io/singularity/

[6] 清水隆博, 河野真治.GearsOS のメタ計算. 琉球大学工学部情報工学科令和 3 年度学位論文 (修士),2021.

[7] Russ Cox, M Frans Kaashoek, and Robert Morris. Xv6, a simple unix-like teaching operating system,2012.

[8] https://www.raspberrypi.org

[9] https://wiki.osdev.org/UEFI

謝辞

感謝します。

2021 年 2 月
奥田 光希