平成29年度 卒業論文

GearsOS on Raspberry Pi

琉球大学工学部情報工学科

145759E 桃原 優

指導教員 河野 真治

目 次

図 目 次

表 目 次

第1章　RaspberryPi上でのGearsOS

当研究室では CbC(Continuation base C) と CbC を用いて実装する GearsOS の研究を行っている。

CbC は Code Segment と Data Segment という単位でプログラムを記述する。Code Segment は並列処理の単位として利用でき、Data Segment はデータそのもので型を持っていて、CbC はメタレベルの処理，並列処理を記述することができる。

メタレベルの処理では、メモリ管理、スレッド管理、CPU や GPU の資源管理を記述することができる。

本研究では、ARM で動くシングルボードコンピュータである Raspberry Pi 上で Gears OS を動かせるようになる事で、ハードウェア上でもメタレベルの処理、並列実行ができるプログラミングを記述できるようになる事を目指している。

しかし、メモリの関係上 RaspberryPi 上で CbC の make を行うと、かなりの時間がかかる。

解決案として、OSX 上で CrossCompile を行う方法と、Linux 環境で CrossCompile を行う方法を提案する。

第2章　RaspberryPi上での実装

2.1　CbCのmake時間

Raspberry Pi 1 のメモリは 256MB と小さいため、 CbC を make することができない。Raspberry Pi 3 だとメモリは 1GB あり CbC を make できるが、時間がかかる。

make 時間の比較として研究室のサーバでメモリ 16GB の firefly と学科のサーバの一つで Linux 環境であるメモリ 756GB の DALMORE を用いる。なお、firefly と DALMORE では Google によって開発された build システムの ninja-build を用いて make を行なった。その結果を表 2.1 に示す。

環境 (メモリ)	時間	比較
RaspberryPi(1GB)	15 時間 11 分 06 秒	1.00 倍
OSX(16GB)	2 時間 16 分 06 秒	6.69 倍
Linux(756GB)	2 分 26 秒	374.42 倍

表 2.1: make 時間の比較

第3章　CrossCompile

3.1　CrossCompile

CrossCompile を行うことで make 時間の問題を解決する方法がある。CrossCompile とは、別の OS で実行可能なコードを生成するコンパイルの手法である。Raspbian 以外の OS 環境であらかじめ Raspberry Pi で CbC が動くように CrossCompile を行い、そのコードを Raspberry Pi に移す事で、実行できるようになる。

Raspberry Pi の OS である Rasbian は qemu によるメモリの拡張もできないので、別の手法で Raspberry Pi 上に CbC を実装する方が好ましい。

3.2　xv6

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 に書き換え、x86 に移植した Xv6 OS である。Xv6 は Raspberry Pi に移植する事ができる。ANSI-C で書かれている Xv6 を CbC に書き直す事で、Raspberry Pi で CbC を動かせるようになる。

3.3　T 図形

CbC を Raspberry Pi で動かすためのアプローチの手法を、I と T の形をした図の組み合わせによって説明を行う。I の上部分に cbclang や Xv6 などのソースコード名を、下部分にその機能の記述言語を記してある。T の下にある I は特別で、上に VM 下に VM を乗せている OS が記されている。 T の上部分は左に入力されるファイル、右に出力されるその機能によって出力されるファイルが記され、下部分にその機能の記述言語が記されている。

例として、cbclang のソースコード (I) と、 Raspberry Pi 上の clang (T) を図 3.1 に示す。

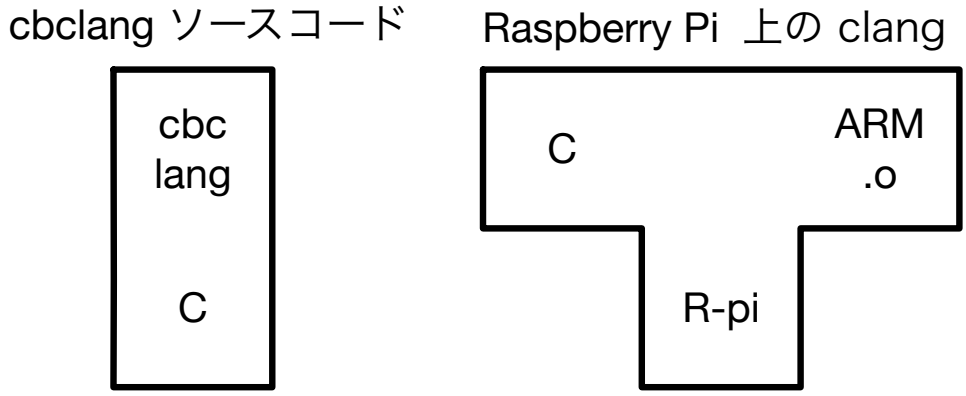

図 3.1: T 図形の例

Raspberry Pi は ARM のコードを生成する。CbC は C によって記述されているため、C から ARM.o を生成し、ARM.o から a.out を出力する。Raspberry Pi 上で xv6 を実装し、cbc から a.out を出力するまでの過程を図 3.2 に示す。

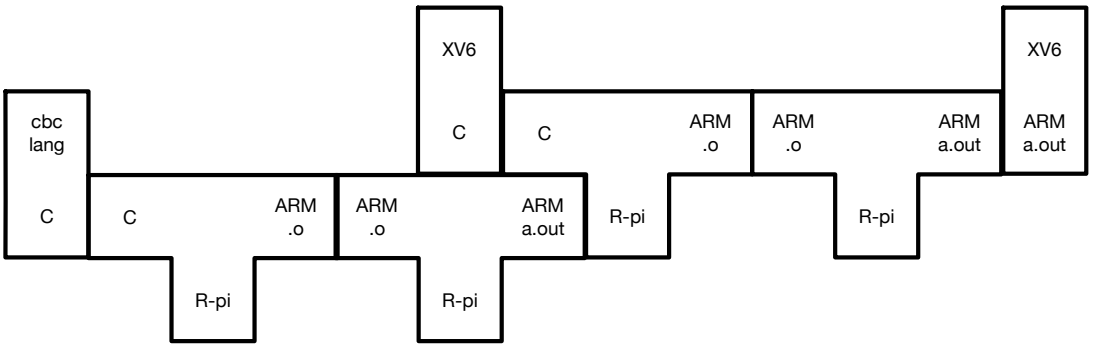

図 3.2: Raspberry Pi

第4章　OSX

4.1　LLVM

arm-linux-gnueabihf-gcc というツールチェーンを使用し、C で書かれたファイルを Cross-Compile することで OSX で ARM のコードを生成できる事を確認した。その際のコマンドをリスト 4.1 に、生成されたファイルに file コマンドで調べた結果をリスト 4.2 に示す。

```
1 /usr/local/linaro/arm-linux-gnueabihf-raspbian/bin/arm-linux-gnueabihf-gcc -g -o
      hello hello.c
```

Code 4.1: CrossCompile の例

```
1 $ file hello
2 hello: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked,
      interpreter
3 /lib/ld-linux-armhf.so.3, for GNU/Linux 2.6.26,
4 BuildID[sha1]=a78af9ba00197d52a8ed7cbac433b24360c57283, not stripped
```

Code 4.2: 生成したファイル

このファイルは Raspberry Pi 上で実行する事ができる。

4.2　GCC

4.3 xv6

OSX 上で xv6 を実装する場合、C から mach-o のバイナリファイルが生成される。この時、出力は mach-o になるので、mach-o の loader 作ることで CbC が動かせるようになる。

OSX 上で行えるためコンパイルの速度向上が望める。xv6 を実装し CbC から a.out を出力するまでの過程を、図 4.1 に示す。

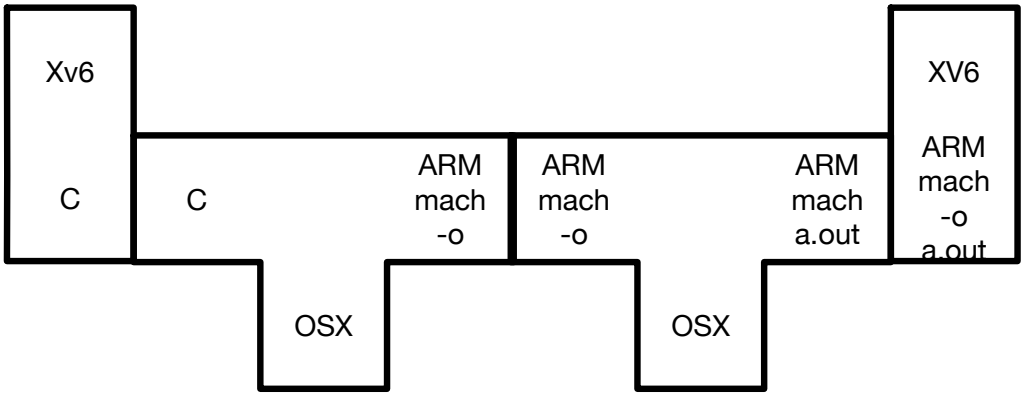

図 4.1: OSX

第5章　Linux

5.1　LLVM

Linux の LLVM でコンパイルすることができれば elf のコードを書けるようになるので、elf の loader を作る事で、CbC を動かすことができる。

C 言語で記述されたファイルから RaspberryPi で実行できるファイルを実行できるように CrossCompile を行なった際のコマンドを Code5.1 に示す。

```
1  ./bin/clang -target arm-linux-gnueabi -mfpu=neon-fp16 -marm -I /net/open/Linux/arm/
       gcc-arm-none-eabi-7-2017-q4-major/arm-none-eabi/include/ hello.c -c -mfloat-abi
       =hard
```

Code 5.1: Linux 上で C 言語ファイルの CrossCompile

コマンドの説明は次のようになる。

- arm-linux-gnueabi は apt でインストールでき、/net/open/Linux/arm/ に展開される。-I オプションでそのディレクトリを読み込む。
- target オプションで arm-linux-gnueabi を指定することで、arm のコードを生成している。
- mfpu=neon-fp16 は ARMv7 の浮動小数点を有効にしている。
- marm で ARM の命令セットをターゲットにしている。
- mfloat-abi=hard は浮動小数点演算にハードウェア命令を使用する

また、make する際に、Cross 環境を整える事で Compile する度に Code5.1 のようなコマンドを省略することができる。Makefile の一部を Code5.2 に示す。

```
1  QEMU = qemu-system-arm
2
3 include makefile.inc
4 CC = /usr/local/cbclang/bin/clang
5 AS = arm-linux-gnu-as
6 LD = arm-linux-gnu-ld
7 OBJCOPY = arm-linux-gnu-objcopy
8 OBJDUMP = arm-linux-gnu-objdump
9
10 CFLAGS = -target armv6-arm-none-eabi -fno-pic -static -fno-builtin -fno-strict-
       aliasing -Wall -I. -I ../cbclang/arm -g -O0
11
12
13 ASFLAGS = -target armv6-arm-none-eabi
14
15 LIBGCC = $(shell $(CC) $(CFLAGS) -print-libgcc-file-name)
```

Code 5.2: Linux 上の LLVM make ファイル

- CC はコンパイラで、すでに make した CbC のディレクトリを指定している。
- AS はアセンブラ、LD はリンカーで、それぞれ arm-linux-gnu を指定してあげればよい。
- CFLAGS
- ASFLAGS

5.2 GCC

Linux 用の gcc を CbC に書き直す際に、gcc7 に書き直せば linker がそのまま使えるので、xv6 で動くようになる。

```
1  QEMU = qemu-system-arm
2
3  include makefile.inc
4  CC = arm-linux-gnu-gcc
5  AS = arm-linux-gnu-as
6  LD = arm-linux-gnu-ld
7  OBJCOPY = arm-linux-gnu-objcopy
8  OBJDUMP = arm-linux-gnu-objdump
9
10 CFLAGS = -fno-pic -static -fno-builtin -fno-strict-aliasing -Wall -I. -g -O0
11
12 ASFLAGS =
13
14 LIBGCC = $(shell $(CC) -print-libgcc-file-name)
15
16 LINK_BIN = $(call quiet-command,$(LD) $(LDFLAGS) \
17        -T $(1) -o $(2) $(3) $(LIBS) -b binary $(4), "␣␣LINK␣␣␣␣␣$(TARGET_DIR)$@")
18
19 LINK_INIT = $(call quiet-command,$(LD) $(LDFLAGS) \
20        $(1) -o $@.out $<, "␣␣LINK␣␣␣␣␣$(TARGET_DIR)$@")
21 OBJCOPY_INIT = $(call quiet-command,$(OBJCOPY) \
22        -S -O binary --prefix-symbols="_binary_$@" $@.out $@, "␣␣OBJCOPY␣␣$(
             TARGET_DIR)$@")
```

Code 5.3: Linux での GCC CrossCompile

- CC はコンパイラで、すでに make した CbC のディレクトリを指定している。
- AS はアセンブラ、LD はリンカーで、それぞれ arm-linux-gnu を指定してあげればよい。
- CFLAGS
- ASFLAGS

5.3 xv6

メモリを割く事で Linux 上でもコンパイルの速度向上が望める。CbC は C によって記述されているので、C から ARM.o のバイナリを生成し、ARM.o から a.out を出力する。OSX 上に立ち上げた Linux 上で xv6 を実装するまでの過程を、図 5.1 に示す。

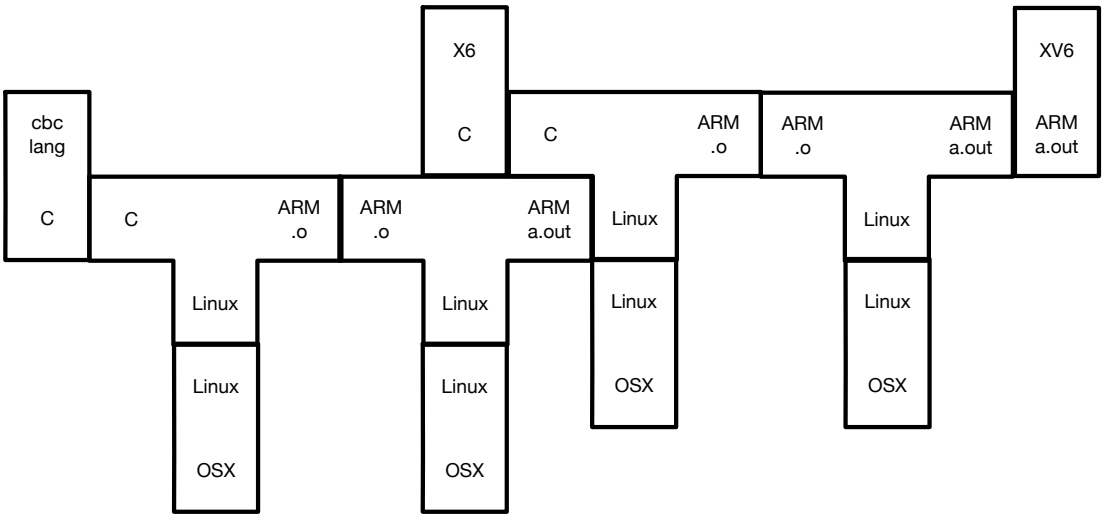

図 5.1: OSX Linux VM

第6章　今後の課題

Xv6 で CbC が動くようになれば、Raspberry Pi 以外のハードウェアでの実装も容易になるので、Linux 上での実装を目指して研究を進めていく。xv6 で CbC が動けば、続けて Linux 上で Gears OS の実装も行なっていく。

謝辞

参考文献

付録