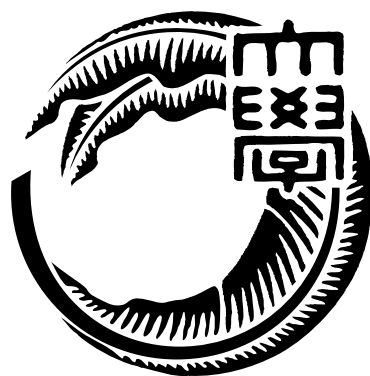


令和2年度 卒業論文

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

第 1 章	初めに	1
1.1	要旨	1
1.2	論文の構成	1
第 2 章	Continuation based C(CbC)	2
2.1	Continuation based C(CbC)	2
2.1.1	GCC	2
2.1.2	LLVM/Clang	3
2.1.3	CrossCompile	3
2.1.4	Singularity	3
2.2	CbC on GCC CrossCompile	3
第 3 章	Raspberry Pi 上の GearsOS	6
3.1	GearsOS	6
3.2	xv6	6
3.3	Raspberry Pi	6
3.4	Raspberry Pi 上の xv6	6
第 4 章	UEFI	8
4.1	UEFI	8
4.2	BIOS と UEFI	8
4.2.1	CPU	8
4.2.2	メモリ空間	9
4.2.3	BOOT	9
4.2.4	デバイス	9
4.3	UEFI Application	9
4.4	UEFI 開発環境	10
4.5	Raspberry Pi 上の UEFI	10
4.6	QEMU 上の UEFI	10
第 5 章	Boot Loader	11
5.1	Boot Loader	11
第 6 章	今後の課題	12
6.1	今後の課題	12

目 次

2.1	CbC の遷移	2
3.1	Raspberry Pi と USBserial 接続	7
4.1	UEFI Application	10
5.1	bootloader	11

表 目 次

4.1 BIOS と UEFI の違い	8
-------------------------------	---

ソースコード目次

2.1	CbC_gcc_cross.def	3
2.2	singularity build	4
2.3	singularity 上で CrossCompile	4
2.4	hello.cbc	5
3.1	screen コマンド	7

第1章 初めに

1.1 要旨

2017年、Intel社は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。問題点の一つとして、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、当研究室で開発を行っ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。

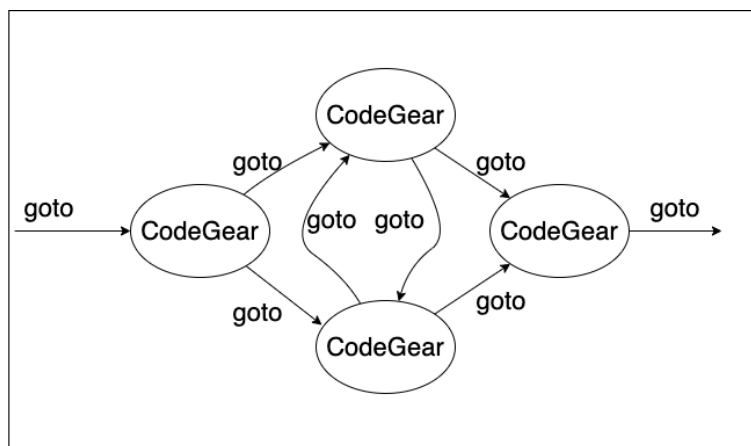


図 2.1: CbC の遷移

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。Raspberry 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、実行できるようになる。

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ソースコード 2.1 を作成する。

ソースコード 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/buildddir
30     cd /CbC_gcc/buildddir
31     sh /CbC_gcc/configure -v --prefix=/usr/local/cbc_gcc --target=arm-linux-
32     gnueabihf \
33     --enable-languages=c,lto --with-arch=armv7-a --with-fpu=vfpv3-d16 --with-
34     float=hard \
35     --disable-multilib --disable-nls --enable-checking=tree,rtl,assert,types "
36     CFLAGS=-g3_00" \
37     --disable-libstdcxx --disable-libssp --disable-libstdcxx-pch --disable-
38     libmudflap \
39     --with-newlib --with-sysroot=/ --with-as=/usr/arm-linux-gnueabihf/bin/as
40     --with-ld=/usr/arm-linux-gnueabihf/bin/ld
41     ./config.status
42     make -j$(nproc) all-gcc
43     make install-gcc
44
45     # setup linker
46     ln -s /usr/arm-linux-gnueabihf/lib/crt*.o /usr/local/cbc_gcc/lib/gcc/arm-
47     linux-gnueabihf/10.0.1/ && \
48     ln -s /usr/lib/gcc-cross/arm-linux-gnueabihf/9/crt*.o /usr/local/cbc_gcc/lib/
49     gcc/arm-linux-gnueabihf/10.0.1/ && \
50     ln -s /usr/lib/gcc-cross/arm-linux-gnueabihf/9/libgcc* /usr/local/cbc_gcc/lib
51     /gcc/arm-linux-gnueabihf/10.0.1/
52
53     # download arm-none-eabi
54     wget "https://developer.arm.com/-/media/Files/downloads/gnu-rm/10-2020q4/gcc-
55     arm-none-eabi-10-2020-q4-major-x86_64-linux.tar.bz2?revision=ca0cbf9c-9
56     de2-491c-ac48-898b5bbc0443&la=en&hash=68760
57     A8AE66026BCF99F05AC017A6A50C6FD832A" -O /tmp/arm.tar.bz2 && \
58     tar -jxvf /tmp/arm.tar.bz2 -C /opt && \
59     mv /opt/gcc-arm-none-eabi-10-2020-q4-major /opt/tools
60
61     %environment
62     export LANG=C
63     export PATH=/usr/local/cbc_gcc/bin:$PATH

```

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

ソースコード 2.2: singularity build

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

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

ソースコード 2.3: singularity 上で CrossCompile

```
1 singularity shell cbc_gcc_cross.sif
2 Singularity> arm-linux-gnueabi-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 が ARM Platform の 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を行っているsOS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の個々のシステムコールが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ようになる。

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である。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1GB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 側でソースコード 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ようになる。この時、ソースコード 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16EB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。起動したら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

UEFIの内部構造は図4.1になっている。赤い点線で囲まれている部分がUEFIのApplicationの部分になる。UEFI Boot Managreがロード、実行するプログラムのことをUEFI Applicationと呼ぶ。UEFI仕様に適合するUEFI実装はEFI Byte Codeを含んでいるため全てのプロセッサ、アーキテクチャと互換性がある。つまりUEFI Applicationにも互換性がある。また、UEFI ApplicationはC言語で記述することが可能なので、個人で開

発することが可能である。UEFI Application は efi という拡張子で C 言語で書かれたコードをコンパイルすると UEFI Application になる。UEFI OS Loader は UEFI Application の一種であり、これを使えば bootloader も開発が可能になる。

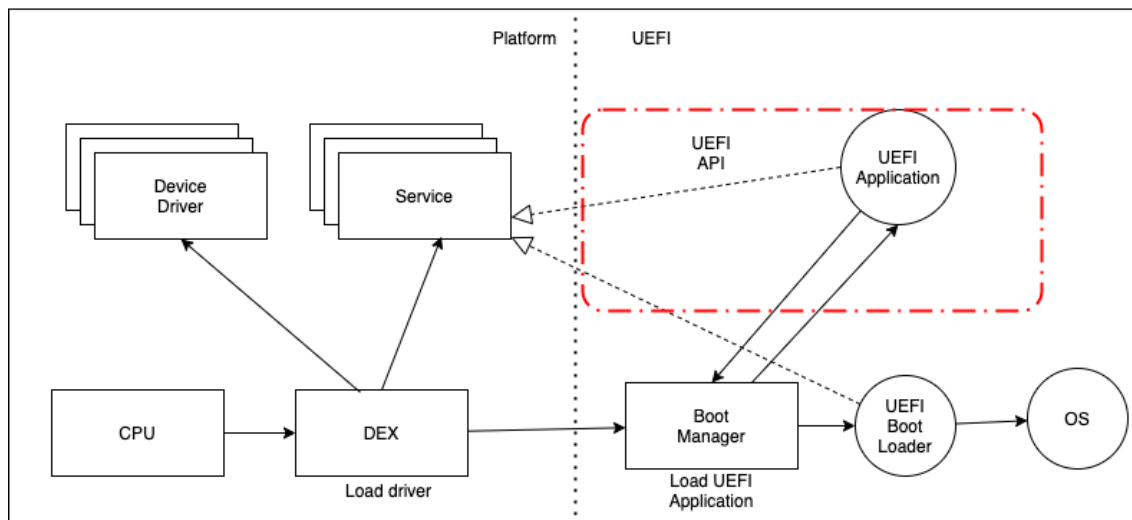


図 4.1: UEFI Application

4.4 UEFI 開発環境

UEFIを開発するにあたって使われる開発環境が2つ存在する。Intelのtianocore EDK2[10]とGNU-efi[11]である。Tianocoreは、独自のビルドシステムをもつ大規模で複雑な環境で、CrossCompilerとしてGCC、MinGW、Microsoft Visual C++などが使用できる。UEFI ApplicationのコンパイルだけでなくBIOS ROMにフラッシュするUEFIファームウェアのコンパイルにも利用できる。Gnu-efiは、システムのネイティブGCCでUEFIアプリケーションをコンパイルするためのライブラリとヘッダのセットである。UEFIファームウェアのコンパイルには使用できない。UEFI Applicationをリンクするためのライブラリがいくつかあり、UEFI Applicationの開発に特化している。

4.5 Raspberry Pi上のUEFI

4.6 QEMU上のUEFI

第5章 Boot Loader

5.1 Boot Loader

Boot Loader は、OS を Load して Boot させる役割をもつプログラムである。図 5.1 は、UEFI と Platform の関係を表したもので、赤い円の部分が BootLoader である。図 5.1 のように BootLoader は BootManager から起動され、OS を Load し、boot させる。

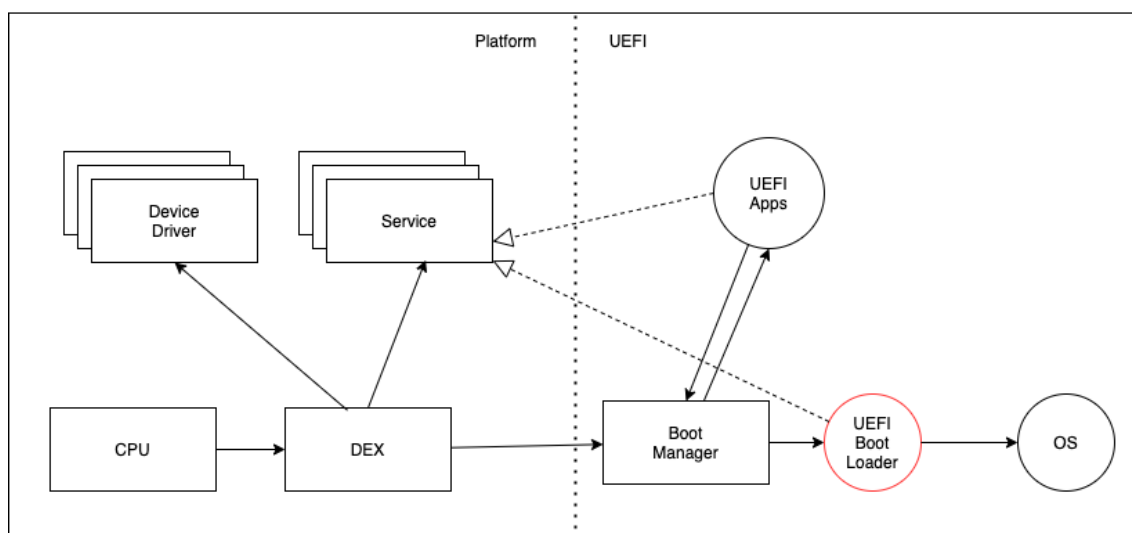


図 5.1: bootloader

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] 清水隆博, 河野慎治..xv6 の構成要素の継続の分析. 情報処理学会研究報告,2020.
- [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] 東恩納 琢偉, 河野真治.Gears OS でモデル検査を実現する手法について. 情報処理学会研究報告 2020
- [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>
- [10] <https://github.com/tianocore/tianocore.github.io/wiki/EDK-II>
- [11] <https://sourceforge.net/projects/gnu-efi>

謝辞

本研究の遂行及び本論文の作成にあたり御多忙にもかかわらず終始懇切丁寧に御指導と御教授を賜りました河野真治准教授に深く感謝します。本研究の遂行及び本論文の作成にあたり数々の貴重な御助言と細やかな御配慮を戴いた並列信頼研究室の清水隆博さん、東恩納琢偉さん、一木貴裕君さん、安田亮君さん、城後 明慈さんに深く感謝します。また一年間共に研究を行い、数々の御助言並びに細かな御配慮を戴いた宮平賢君、上地悠斗君、大蔵海人君並びに並列信頼研究室の皆様と毎週ゼミでお世話になったハリーズさんに深く感謝します。

2021年2月
奥田 光希