修士 (工学) 学位論文
Master's Thesis of Engineering

継続を基本とした言語による OS のモジュール化

2019 年 3 月
March 2019

宮城 光希
Mitsuki MIYAGI

琉球大学
大学院理工学研究科
情報工学専攻

Information Engineering Course
Graduate School of Engineering and Science
University of the Ryukyus

指導教員 : 教授　和田 知久
Supervisor: Prof. Tomohisa WADA

本論文は、修士 (工学) の学位論文として適切であると認める。

論　文　審　査　会

印

(主　査)　　和田　知久

印

(副　査)　　岡﨑　威生

印

(副　査)　　赤嶺　有平

印

(副　査)　　河野　真治

要旨

現代の OS では拡張性と信頼性を両立させることが要求されている。信頼性をノーマルレベルの計算に対して保証し、拡張性をメタレベルの計算で実現することを目標に Gears OS を設計中である。

Gears OS は Continuation based C (CbC) によってアプリケーションと OS そのものを記述する。OS の下ではプログラムの記述は通常の処理の他に、メモリ管理、スレッドの待ち合わせやネットワークの管理、エラーハンドリング等の記述しなければならない処理が存在する。これらの計算をメタ計算と呼ぶ。メタ計算を通常の計算から切り離して記述するために、Code Gear、Data Gear という単位を提案している。CbC はこの Code Gear と Data Gear の単位でプログラムを記述する。

OS は時代とともに複雑さが増し、OS の信頼性を保証することは難しい。そこで、基本的な OS の機能を揃えたシンプルな OS である xv6 を CbC で書き換え、OS の機能を保証する。

Code Gear は goto による継続で処理を表すことができる。これにより、状態遷移による OS の記述が可能となり、複雑な OS のモデル検査を可能とする。また、CbC は 定理証明支援系 Agda に置き換えることができるように構築されている。これらを用いて OS の信頼性を保証したい。

CbC でシステムやアプリケーションを記述するためには、Code Gear と Data Gear を柔軟に再利用する必要がある。このときに機能を接続する API と実装の分離が可能であることが望ましい。Gears OS の信頼性を保証するために、形式化されたモジュールシステムを提供する必要がある。

本論文では、Interface を用いたモジュールシステムの説明と、xv6 の CbC 書き換えについての考察を行う。

Abstract

Contemporary OS is required to achieve both scalability and reliability. We are designing Gears OS with the goal of guaranteeing reliability for normal level computation and realizing extensibility with meta level computation.

Gears OS describes applications and OS themselves with Continuation based C (CbC). In addition to normal processing, the description of the program under the OS includes memory management, queuing of threads, management of the network, There is a process that must be described such as error handling. These computation are called meta computation. In order to describe the meta computation separately from normal computation, we propose a unit called Code Gear, Data Gear. CbC describes the program in units of Code Gear and Data Gear.

It is necessary to flexibly reuse Code Gear and Data Gear to describe systems and applications. At this time it is desirable that it is possible to separate API and implementation connecting functions. In order to guarantee the reliability of the Gears OS, it is necessary to provide a formatted module system.

In this paper, we describe the module system using Interface, In order to enable meta computation and parallel execution on hardware, We also consider the implementation of Gears OS on Raspberry Pi.

In order to meta computation on hardware and execute it in parallel, basic OS functions are aligned, It is realized by replacing simple xv 6 with Gears OS.

研究関連業績

1. 宮城光希, 河野真治. Code Gear と Data Gear を持つ Gears OS の設計. 第 59 回プログラミング・シンポジウム, Jan, 2018
2. 宮城光希, 桃原 優, 河野真治. GearsOS のモジュール化と並列 API. 情報処理学会システムソフトウェアとオペレーティング・システム研究会 (OS), May, 2018
3. 宮城光希, 河野真治. 継続を中心とした言語 Gears OS のデモンストレーション. 第 60 回プログラミング・シンポジウム, Jan, 2019

目 次

研究関連論文業績 **iii**

第 1 章 OS の拡張性と信頼性の両立 **7**

第 2 章 Gears におけるメタ計算 **10**
2.1 Continuation based C 11
2.2 Code Gear 11
2.3 Data Gear 12
2.4 Meta Code Gear、Meta Data Gear 13
2.4.1 Context と stub Code Gear 14
2.4.2 Meta Gear を用いたメタレベルの処理 15

第 3 章 Gears OS の構成 **17**
3.1 Context 18
3.2 TaskManager 20
3.3 TaskQueue 21
3.4 Workers 21

第 4 章 Interface **22**
4.1 Interface の定義 23
4.2 Interface の実装の記述 24

第 5 章 メタレベルのコードの自動生成 **26**
5.1 Meta Code Gear の生成 26
5.2 Context の生成 28

第 6 章 xv6 の CbC への書き換え **31**
6.1 xv6 のモジュール化 31
6.2 xv6 の構成要素 33
6.2.1 システムコール 33
6.2.2 プロセス 34

6.2.3 ファイルディスクリプタ . . . 34
6.2.4 ファイルシステム . . . 34
6.3 xv6-rpi の CbC 対応 . . . 35
6.4 Raspberry Pi . . . 35
6.5 CbC によるシステムコールの書き換え . . . 35

第 7 章 評価 **44**
7.1 Gears OS のモジュール化 . . . 44
7.2 Code Gear の変換と Meta Gear の自動生成 . . . 45
7.3 xv6 の CbC 書き換えのための環境構築 . . . 45
7.4 xv6 の CbC 書き換え . . . 46

第 8 章 結論 **47**

謝辞 **48**

参考文献 **50**

付録 **51**

図 目 次

1.1 証明とモデル検査による OS の検証 8
1.2 Gears のメタ計算 9

2.1 goto による code gear 間の継続 11
2.2 CodeGear と DataGear 12
2.3 ノーマルレベルの Code Gear の継続 14
2.4 メタレベルの Code Gear の継続 14
2.5 Context が持つ Data Gear へのアクセス 15

3.1 Gears OS の構成図 17

4.1 Stack の Interface とその実装 22

5.1 generate_context による Context の生成 29

6.1 モジュール化した xv6 の構成 32
6.2 read システムコールの遷移図 36

7.1 read システムコールの遷移図 46

表 目 次

ソースコード目次

2.1 code segment の軽量継続 12
2.2 Gears での Stack pop 13
2.3 code meta 15
3.1 Context 18
3.2 enum で定義された Code Gear の番号 20
3.3 Context の初期化 20
3.4 enum で定義された Data Gear の番号 20
3.5 TaskManager の Interface 20
4.1 pushSingleLinkedStack 23
4.2 Stack の Interface 24
4.3 SingleLinkedStack の実装 24
4.4 SingleLinkedStack のデータ構造 25
5.1 変換前の Stack pop 26
5.2 変換後の Stack pop 27
5.3 Gearef、GearImpl 27
5.4 マクロを用いない stub Code Gear 28
5.5 生成された target-context.c 29
6.1 xv6 のシステムコールのリスト 33
6.2 syscall() 36
6.3 cbc_read システムコール 37
6.4 read システムコール 37
6.5 fileread の CbC 書き換えの例 37
6.6 書き換え前の fileread 38
6.7 readi の CbC 書き換えの例 39
6.8 書き換え前の readi 39
6.9 consoleinit 40
6.10 consoleread の CbC 書き換えの例 40
6.11 書き換え前の consoleread 42
7.1 Interface を用いない Gears 44

7.2 Interface を用いた Gears 45

第1章　OS の拡張性と信頼性の両立

コンピュータには CPU、ディスプレイ、キーボードやマウス、ハードディスクなど様々な機器が接続されている。プログラムの処理を行うとき、これらの様々なデバイスのアクセスや資源管理は複雑で容易ではない。異なるハードウェアを扱う際にはそれぞれに対応したプログラミングを行う必要がある。OS とはこれらのデバイスの抽象化や資源管理を行う。

ユーザーは OS のおかげで異なるハードウェアの違いを意識することなくプログラミングをすることができる。例えば同じプログラムで、異なる入力デバイスによる操作や、ディスプレイでの表示などは、それぞれデバイスへのアクセスなどの複雑な処理を OS が隠すことによって可能となっている。

CPU、メモリ、ディスク、などの本来ユーザーが意識しなければならない資源も、OS が資源管理を行うことで意識することなくプログラミングすることができる。

1950 年代におけるコンピューターがプログラムを実行する際には専門のオペレータが存在し、オペレーターがジョブの管理、コンパイラの選択などを行なっていた。しかし後に、ジョブの自動実行、やコンパイラのロードを行うプログラムができた。これが OS の祖先である。コンピュータには科学技術用のコンピューターと商用のコンピューターが開発されていたが、後に汎用コンピューターである System/360 が開発される。その OS/360 は強力なソフトウェア互換を持っていた。ソフトウェア互換により全てのソフトウェアが全てのマシンで動作するようになった。その後、OS は、スプーリングやタイムシェアリングといった機能も導入されるようになった。1970 年代には UNIX が開発され、後に様々なバージョンが開発され、System V、BSD といった派生 OS なども開発され、現在に至る。[1]

OS はさまざまなコンピュータの信頼性の基本である。OS の信頼性を保証する事自体が難しいが、時代とともに進歩するハードウェア、サービスに対応して OS 自体が拡張される必要がある。OS は非決定的な実行を持ち、その信頼性を保証するには、従来のテストとデバッグでは不十分であり、テストしきれない部分が残ってしまう。これに対処するためには、証明を用いる方法とプログラムの可能な実行をすべて数え上げるモデル検査を用いる方法がある。モデル検査は無限の状態ではなくても巨大な状態を調べることになり、状態を有限に制限したり状態を抽象化したりする方法が用いられている。(図 1.1)

証明やモデル検査を用いて OS を検証する方法はさまざまなものが検討されている。検

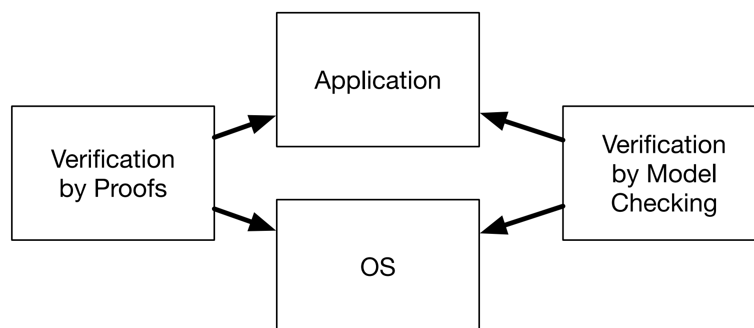

図 1.1: 証明とモデル検査による OS の検証

証は一度ですむものではなく、アプリケーションやサービス、デバイスが新しくなることに検証をやり直す必要がある。つまり信頼性と拡張性を両立させることが重要である。

コンピュータの計算はプログラミング言語で記述されるが、その言語の実行は操作的意味論の定義などで規定される。プログラミング言語で記述される部分をノーマルレベルの計算と呼ぶ。実際にコードが実行される時には、処理系の詳細や使用する資源、あるいは、コードの仕様や型などの言語以外の部分が服属する。これをメタレベルの計算という。この二つのレベルを同じ言語で記述できるようにして、ノーマルレベルの計算の信頼性をメタレベルから保証できるようにしたい。ノーマルレベルでの正当性を保証しつつ、新しく付け加えられたデバイスやプログラムを含む正当性を検証したい。

ノーマルレベルとメタレベルを共通して表現できる言語として Continuation based C(以下 CbC)[2] を用いる。CbC は関数呼出時の暗黙の環境 (Environment) を使わずに、コードの単位を行き来できる引数付き goto 文 (parametarized goto) を持つ C と互換性のある言語である。これを用いて、Code Gear と Data Gear、さらにそのメタ構造を導入する。これらを用いて、検証された Gears OS[3] を構築したい。図 1.2。

Meta Gear を入れかえることにより、ノーマルレベルの Gear をモデル検査することができるようにしたい。ノーマルレベルでの Code Gear と Data Gear は継続を基本とした関数型プログラミング的な記述に対応する。この記述を定理証明支援系である Agda を用いて直接に証明できるようにしたい。Gears OS の記述はそのまま Agda に落ちる。Code Gear、Data Gear を用いた言語である CbC で記述することによって検証された OS を実装することができる。

従来の研究でメタ計算を用いる場合は、関数型言語では Monad を用いる [4]。これは Haskell では実行時の環境を記述する構文として使われている。OS の研究では、メタ計算の記述に型つきアセンブラ (Typed Assembler) を用いることがある [5]。Python や Haskell による記述をノーマルレベルとして採用した OS の検証の研究もある [6, ?]。SMIT などのモデル検査を OS の検証に用いた例もある [7]。

本研究で用いる Meta Gear は制限された Monad に相当し、型つきアセンブラよりは大きな表現単位を提供する。Haskell などの関数型プログラミング言語では実行環境が複雑であり、実行時の資源使用を明確にすることができない。CbC はスタック上に隠された環境を持たないので、メタレベルで使用する資源を明確にできるという利点がある。ただし、Gear のプログラミングスタイルは、従来の関数呼出を中心としたものとはかなり異なる。

Gears の記述をモジュール化するために Interface を導入した。これらの機能は Agda [8] 上で継続を用いた関数型プログラムに対応するようになっている。[9] これにより Code Gear、Data Gear の Agda による証明が可能となるように Gears OS の構築を行った。

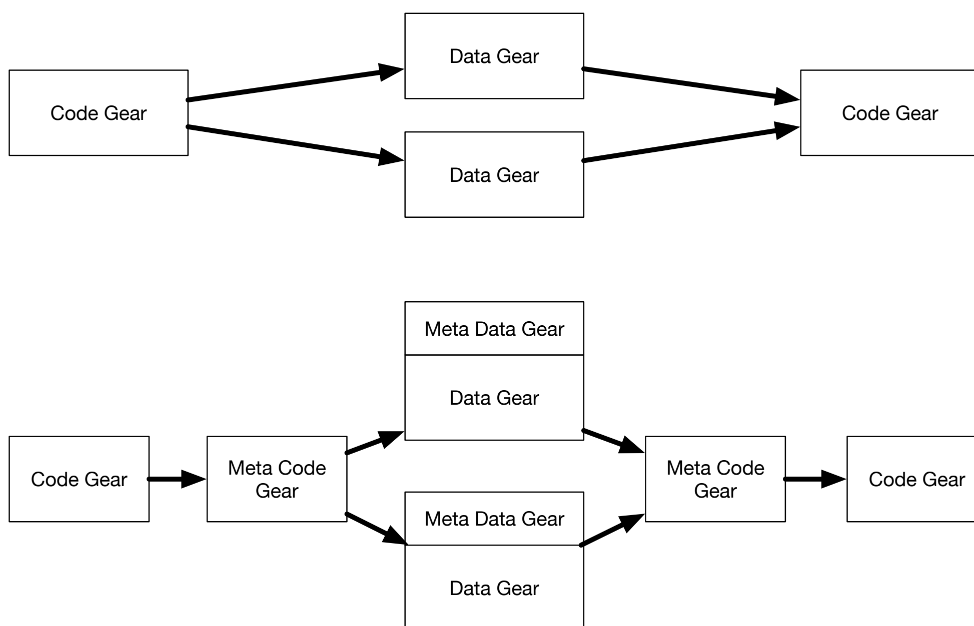

図 1.2: Gears のメタ計算

本論文では、Gears OS の要素である Code Gear、Data Gear、そして、Meta Code Gear、Meta Data Gear の構成を示す。また、OS の信頼性を保証するため、xv6 [10] の CbC による書き換えについて考察する。xv6 は OS の基本的な機能を持っているにも関わらず、Linux などに比べてシンプルであり、CbC に書き換えることによって状態遷移ベースのプログラミングが可能となり、実行可能なプログラムがそのままモデル検査が可能で、Agda による証明が可能な OS となることを目的とする。

第2章　Gears におけるメタ計算

プログラムを記述する際、ノーマルレベルの処理の他に、メモリ管理、スレッド管理、CPU や GPU の資源管理等、記述しなければならない処理が存在する。これらの計算をメタ計算と呼ぶ。

従来の OS では、メタ計算はシステムコールやライブラリーコールの単位で行われる。実行時にメタ計算の変更を行う場合には、OS 内部のパラメータの変更を使用し、実行されるユーザープログラム自体への変更は限定的である。しかし、メタ計算は性能測定、あるいはプログラム検証、さらに並列分散計算のチューニングなど細かい処理が必要で実際のシステムコール単位では不十分である。例えば、モデル検査ではアセンブラあるいはバイトコード、インタプリタレベルでのメタ計算が必要になる。しかし、バイトコードレベルでは粒度が細かすぎて扱いが困難になっている。具体的にはメタ計算の実行時間が大きくなってしまう。

メタ計算を通常の計算から切り離して記述するためには処理を細かく分割する必要がある。しかし、関数やクラスなどの単位は容易に分割できない。そこで当研究室ではメタ計算を柔軟に記述するためのプログラミング言語の単位として Code Gear、Data Gear という単位を提案している。プログラムの処理の単位を Code Gear、データの単位を Data Gear と呼ぶ。これによりシステムコードよりも細かくバイトコードよりも大きなメタ計算の単位を提供できる。

Code Gear、Data Gear にはそれぞれメタレベルの単位である Meta Code Gear、Meta Data Gear が存在し、これらを用いてメタ計算を実現する。

Gears OS は処理やデータの構造が Code Gear、Data Gear に閉じているため、これにより実行時間、メモリ使用量などを予測可能なものにすることが可能になる。

Gears OS には Context と呼ばれる全ての Code Gear、Data Gear のリストを持つ Meta Data Gear が存在する。Gears OS ではこの Context を常に持ち歩いているが、これはノーマルレベルでは見えることはない。

ノーマルレベルの処理とメタレベルを含む処理は同じ動作を行う。しかしメタレベルの計算を含むプログラムとノーマルレベルでは、Data Gear の扱いなどでギャップがある。ノーマルレベルでは Code Gear は Data Gear を引数の集合として引き渡しているが、メタレベルでは Context に格納されており、ここを参照することで Data Gear を扱っている。

このギャップを解消するためにメタレベルでは stub Code Gear と呼ばれる Context から Data Gear の参照を行う Meta Code Gear が Code Gear 継続前に入されこれを解決する。

2.1 Continuation based C

CbC は処理を Code Gear とした単位を用いて記述するプログラミング言語である。Code Gear 間では軽量継続 (goto 文) による遷移を行うので、継続前の Code Gear に戻ることはなく、状態遷移ベースのプログラミングに適している。図 2.1 は Code Gear 間の処理の流れを表している。

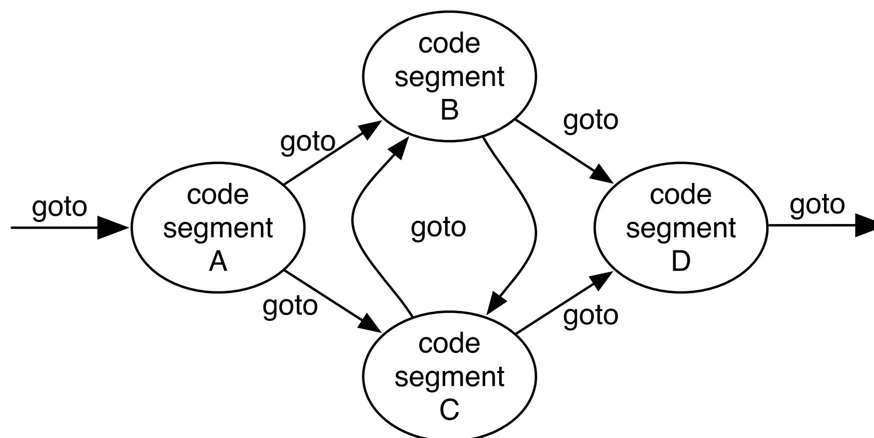

図 2.1: goto による code gear 間の継続

CbC は LLVM[11] と GCC[12] 上で実装されている。Gears OS はこの CbC を用いて記述されている。

2.2 Code Gear

Code Gear は CbC における最も基本的な処理単位である。関数に比べて細かく分割されているのでメタ計算をより柔軟に記述できる。ソースコード 2.1 はCbC における Code Gear の一例である。

ソースコード 2.1: code segment の軽量継続

```
__code cs0(int a, int b){
  goto cs1(a+b);
}

__code cs1(int c){
  goto cs2(c);
}
```

Code Gear は__code Code Gear 名 (引数) の形で記述される。Code Gear は戻り値を持たないので、関数とは異なり return 文は存在しない。次の Code Gear への遷移は goto Code Gear 名 (引数) で次の Code Gear への遷移を記述する。ソースコード 2.1 での goto cs1(a+b); がこれにあたる。この goto の行き先を継続と呼び、このときの a+b が次の Code Gear への出力となる。Scheme の継続と異なり CbC には呼び出し元の環境がないので、この継続は単なる行き先である。したがってこれを軽量継続と呼ぶこともある。cs1 へ継続した後は cs0 へ戻ることはない。軽量継続により、並列化、ループ制御、関数コールとスタックの操作を意識した最適化がソースコードレベルで行えるようにする。

CbC は軽量継続による遷移を行うので、継続前の Code Gear に戻ることはなく、状態遷移ベースのプログラミングに適している。

2.3 Data Gear

Data Gear は Gears OS におけるデータの単位である。Gears OS では Code Gear は Input Data Gear、Output Data Gear を引数に持ち、任意の Input Data Gear を参照し、Output Data Gear を書き出す。図 2.2 Code Gear はこのとき渡された引数の Data Gear 以外を参照することはない。この Data Gear の対応から依存関係の解決を行う。

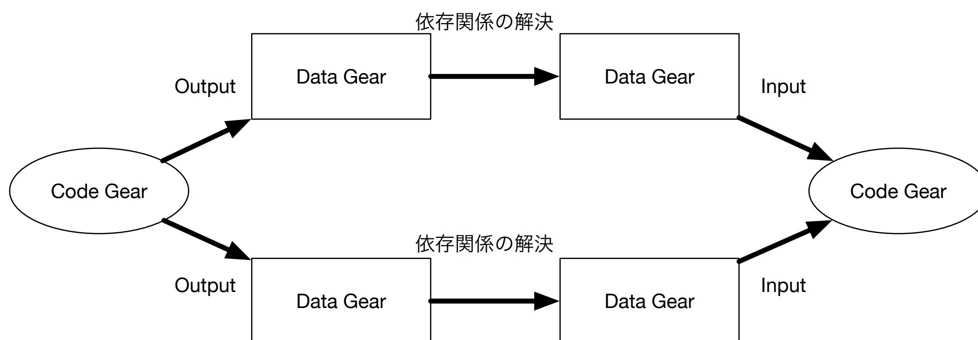

図 2.2: CodeGear と DataGear

ソースコード 2.2 は Gears OS での Stack の pop 操作の Code Gear の例である。popSingleLinkedStack での引数 stack が Input Data Gear、next は継続先の Code Gear を示す。また、next の引数の data が Output Data Gear、... は可変長引数であることを示している。Input Data Gear で受け取った Stack に対して pop の操作を行った後に、取り出したデータを Output Data Gear として書き出し、goto next で引数で受けた次の Code Gear へと継続する。

ソースコード 2.2: Gears での Stack pop

```
1
2  __code stackTest3(struct Stack* stack) {
3      goto stack->pop(assert3);
4  }
5
6  __code popSingleLinkedStack(struct SingleLinkedStack* stack, __code next(
       union Data* data, ...)) {
7      if (stack->top) {
8          data = stack->top->data;
9          stack->top = stack->top->next;
10     } else {
11         data = NULL;
12     }
13     goto next(data, ...);
14 }
15
16 __code assert3(struct Node* node, struct Stack* stack) {
17     assert(node->color == Red);
18     goto exit_code(0);
19 }
```

2.4 Meta Code Gear、Meta Data Gear

Gears OS ではメタ計算 を Meta Code Gear、Meta Data Gear で表現する。CbC での記述はメタ計算を含まないノーマルレベルでの記述と、 Code Gear、Data Gear の記述を含むメタレベルの記述の 2 種類がある。メタレベルでもさらに、メタ計算を用いることが可能になっている。この 2 つのレベルはプログラミング言語レベルでの変換として実現される。メタレベルでの変換系は本論文では、Perl による変換スクリプトにより実装されている。

Gears OS では、Meta Code Gear は通常の Code Gear の直前、直後に挿入され、メタ計算を実行する。Code Gear 間の継続はノーマルレベルでは図 2.3 のように見える。メタレベルでは Code Gear は図 2.4 のように継続を行なっている。

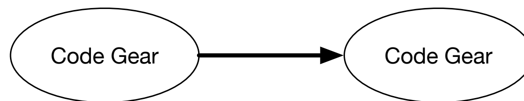

図 2.3: ノーマルレベルの Code Gear の継続

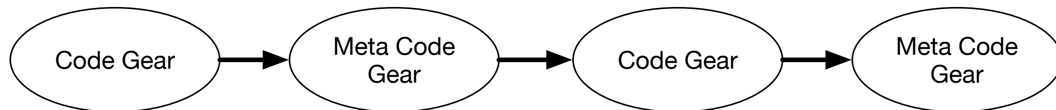

図 2.4: メタレベルの Code Gear の継続

2.4.1 Context と stub Code Gear

Gears OS では Context と呼ばれる、使用されるすべての Code Gear、Data Gear を持つ Meta Data Gear を持っている。Gears OS は必要な Code Gear、Data Gear を参照したい場合、この Context を通す必要がある。しかし Context を通常の計算から直接扱うのはセキュリティ上好ましくない。そこで Context から必要なデータを取り出して Code Gear に接続する Meta Code Gear を定義し、これを介して間接的に必要な Data Gear にアクセスする。この Meta Code Gear を stub Code Gear と呼ぶ。stub Code Gear は Code Gear 毎にあり、次の Code Gear へと継続する前に挿入される。つまり goto による継続を行うと、実際には次の Code Gear の stub Code Gear を呼び出す。stub Code Gear では、継続先が求める Input Code Gear、Output Code Gear を Context から参照している。

図 2.5 はメタレベルで見た Data Gear へのアクセスを図示したものである。メタレベルでは Code Gear は Context が持つ Data Gear へのポインタを渡されており、そこへ Output を行う。stub Code Gear は Meta Data Gear である Context が持つ Data Gear を、Input Data Gear、Output Data Gear として参照し、継続先のノーマルレベルの Code Gear へと遷移する。

次の Code Gear に継続する際にもメタレベルでは Meta Code Gear が挟まれる。ノーマルレベルでは goto codeGear() で次の Code Gear の継続を記述するので、直接 Code Gear へ継続するように見えるが、実際には __code meta へと継続している。__code meta では Context の持つ Code Gear のリストから目的の Code Gear へと継続している。(ソースコード 2.3)

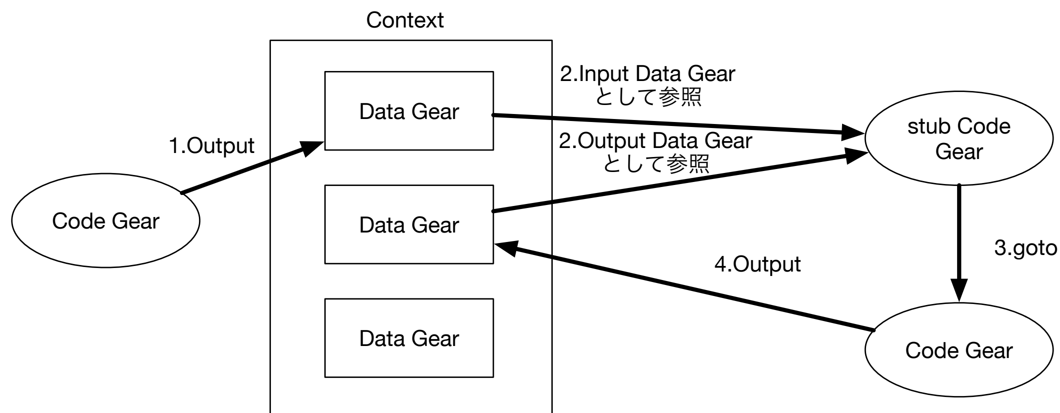

図 2.5: Context が持つ Data Gear へのアクセス

ソースコード 2.3: code meta

```

__code meta(struct Context* context, enum Code next) {
    goto (context->code[next])(context);
}
```

Code Gear と Data Gear は Interface と呼ばれるまとまりとして記述される。Interface は使用される全ての Data Gear の定義と、それに対する操作を行う Code Gear の集合である。Interface 作成時に Code Gear の集合を指定することにより複数の実装を持つことができる。Interface の操作に対応する Code Gear の引数は Interface に定義されている Data Gear を通して指定される。一つの実行スレッド内で使われる Interface の Code Gear と Data Gear は Context に格納される。

Code Gear の継続は関数型プログラミングからみると継続先の Context を含む Closure となっている。これを記述するために継続に不定長引数を追加する構文をスクリプトの変換機能として用意した。メタ計算側ではこれらの Context を常に持ち歩いているので goto 文で引数を用いることはなく、行き先は Code Gear の番号のみで指定される。

これにより Interface 間の呼び出しを C++ のメソッド呼び出しのように記述することができる。Interface の実装は、Context 内に格納されているので、オブジェクトごとに実装を変える多様性を実現できている。

2.4.2 Meta Gear を用いたメタレベルの処理

Context を複製して複数の CPU に割り当てることにより並列実行を可能になる。これによりメタ計算として並列処理を記述したことになる。

Gears のスレッド生成は Agda の関数型プログラミングに対応して行われるのが望ましい。そこで、par goto 構文を導入し、Agda の継続呼び出しに対応させることにした。par goto では Context の複製、入力の同期、タスクスケジューラーへの Context の登録などが行われる。par goto 文の継続として、スレッドの join に相当する __exit を用意した。__exit により par goto で分岐した Code Gear の出力を元のスレッドで受け取ることができる。

関数型プログラムではメモリ管理は GC などを通して暗黙に行われる。Gears OS ではメモリ管理は stub などのメタ計算部分で処理される。例えば、寿命の短いスレッドでは使い捨ての線形アロケーションを用いる。

第3章　Gears OS の構成

Gears OS は Code Gear、 Data Gear の単位を用いて開発されており、CbC で記述されている。Gears OS は Context と呼ばれる使用されるすべての Code Gear、Data Gear 持っている Meta Data Gear を持つ。Gears OS は継続の際この Context を常に持ち歩き、必要な Code Gear、Data Gear を参照したい場合、この Context を通して参照する。

Gears OS は以下の要素で構成される。

・Context
・TaskManager
・TaskQueue
・Workers

図 3.1 に Gears OS の構成図を示す。

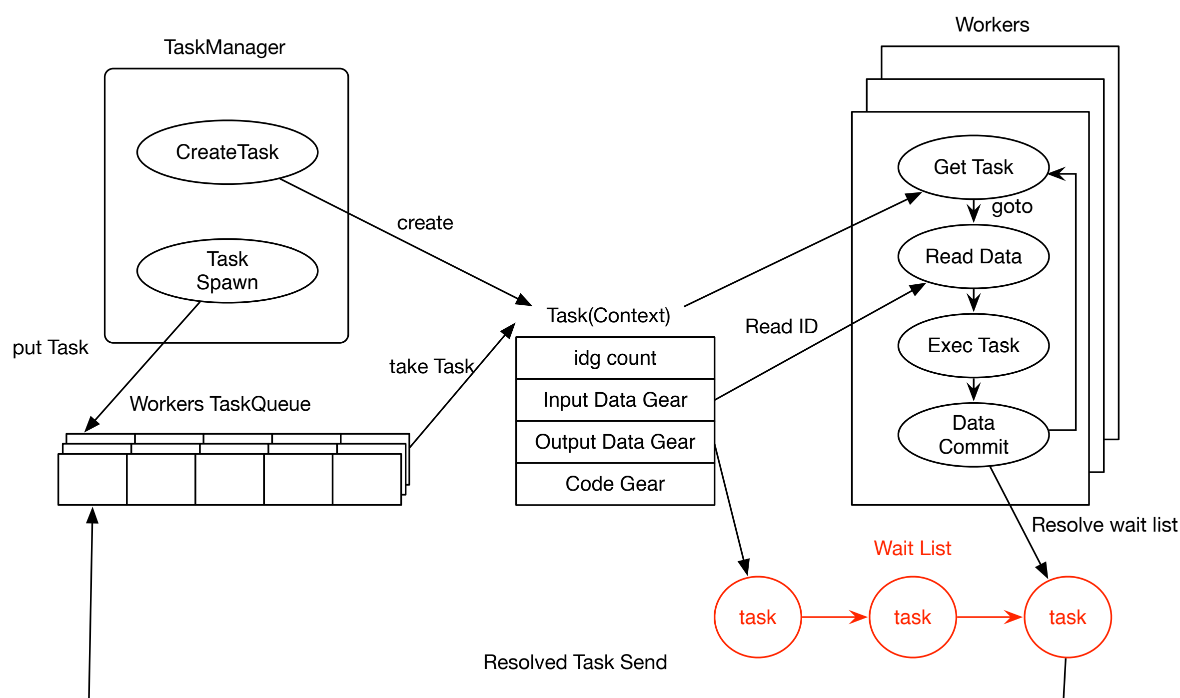

図 3.1: Gears OS の構成図

TaskManager は Task を実行する Worker の生成、管理、Task の送信を行う。Gears OS における Task Queue は Synchronized Queue で実現される。Worker は TaskQueue から Task である Context を取得し、Task の Code Gear を実行し、Output Data Gear の書き出しを行っている。Input/Output Data Gear の依存関係が解決されたものから並列実行される。

TaskManager は Task である Context の生成を行う。実行する Task の Input Data Gear が揃っているなら TaskQueue へ put する。Worker は Task の取得を行い、Task を実行する。Task 実行後 Data Gear の書き出しを行い、再び Task の取得を行う。Data Gear の依存関係が解決されると Task Queue に Task が送り込まれる。

3.1 Context

Context は実行する Code Gear と Data Gear を全て持っている Meta Data Gear である。Context は通常の OS のスレッドに対応する。また Context は並列実行における Task の表現としても用いられる。Context は、ソースコード 3.1 のように定義されている。Context は、実行される Task への Code Gear や、ソースコード 3.1 5、6 行目で記述されているように Code Gear のリスト、Data Gear のリスト、Data Gear を確保するためのメモリ空間を持っている。Context が持つ Data Gear のメモリ空間は事前に確保され、Data Gear のメモリ確保の際に heap の値をずらしメモリを割り当てる。

ソースコード 3.1 13〜30 行目は並列実行用の Task として扱うための情報である。待ち合わせの Input Data Gear のカウンタ、Input/Output Data Gear が格納されている場所を示すインデックス、GPU での実行フラグを持っている。

Gears OS では Data Gear は構造体で定義されている。メタ計算では任意の Data Gear を一律で扱うため、全て union で定義されている。(ソースコード 3.1 33 行目〜) Context には Data Gear の Data Type の情報が格納されている。この情報から確保する Data Gear のサイズなどを決定する。

ソースコード 3.1: Context

```
1  /* Context definition */
2  struct Context {
3      enum Code next;
4      int codeNum;
5      __code (**code) (struct Context*);
6      union Data **data;
7      void* heapStart;
8      void* heap;
9      long heapLimit;
10     int dataNum;
11
12     // task parameter
13     int idgCount; //number of waiting dataGear
```

```
14     int idg;
15     int maxIdg;
16     int odg;
17     int maxOdg;
18     int gpu; // GPU task
19     struct Worker* worker;
20     struct TaskManager* taskManager;
21     struct Context* task;
22     struct Element* taskList;
23 #ifdef USE_CUDAWorker
24     int num_exec;
25     CUmodule module;
26     CUfunction function;
27 #endif
28     /* multi dimension parameter */
29     int iterate;
30     struct Iterator* iterator;
31 };
32 
33 union Data {
34     struct Meta {
35         enum DataType type;
36         long size;
37         long len;
38         struct Queue* wait; // tasks waiting this dataGear
39     } Meta;
40     struct Context Context;
41     // Stack Interface
42     struct Stack {
43         union Data* stack;
44         union Data* data;
45         union Data* data1;
46         enum Code whenEmpty;
47         enum Code clear;
48         enum Code push;
49         enum Code pop;
50         enum Code pop2;
51         enum Code isEmpty;
52         enum Code get;
53         enum Code get2;
54         enum Code next;
55     } Stack;
56     // Stack implementations
57     struct SingleLinkedStack {
58         struct Element* top;
59     } SingleLinkedStack;
60     ....
61 }; // union Data end       this is necessary for context generator
```

Context は、ソースコード 3.2 のように Code Gear の番号を持っており。初期化の際に Code Gear のアドレスと対応付けている。（ソースコード 3.3）

ソースコード 3.2: enum で定義された Code Gear の番号

```
enum Code {
    C_popSingleLinkedStack,
    C_pushSingleLinkedStack,
    C_stackTest3,
    C_assert3,
    ...
};
```

ソースコード 3.3: Context の初期化

```
    context->code[C_popSingleLinkedStack] = popSingleLinkedStack_stub;
    context->code[C_pushSingleLinkedStack]  = pushSingleLinkedStack_stub;
    context->code[C_stackTest3] = stackTest3_stub;
    context->code[C_assert3] = assert3_stub;
```

Data Gear も Code Gear と同様に Context が番号を持っている。(ソースコード 3.4) Context の初期化の際に引数格納用の Data Gear が生成される。この Data Gear は Code Gear が継続する際に、継続先の Code Gear が要求する引数を格納するためのものである。生成された Data Gear は data のリストと番号の対応から参照される。

ソースコード 3.4: enum で定義された Data Gear の番号

```
enum DataType {
    D_Code,
    D_SingleLinkedStack,
    D_Stack,
    D_TaskManager,
    D_Worker,
    ...
};
```

3.2 TaskManager

TaskManager は、CPU、GPU の数に応じた Worker の生成、管理、Task の送信を行う。ソースコード 3.5 は TaskManager の Interface である。

TaskManager は、初期化の際にそれぞれ指定した CPU、GPU の数の Worker を生成する。また、実行する Task の Input Data Gear が用意されているかどうか判断し、全て用意されていた場合、その Task を Worker の Queue に送信する。

ソースコード 3.5: TaskManager の Interface

```
typedef struct TaskManager<Impl>{
    union Data* taskManager;
    struct Context* task;
    struct Element* taskList;
```

```
5     __code spawn(Impl* taskManager, struct Context* task, __code next
    (...));
6     __code spawnTasks(Impl* taskManagerImpl, struct Element* taskList,
    __code next1(...));
7     __code setWaitTask(Impl* taskManagerImpl, struct Context* task,
    __code next(...));
8     __code shutdown(Impl* taskManagerImpl, __code next(...));
9     __code incrementTaskCount(Impl* taskManagerImpl, __code next(...));
10    __code decrementTaskCount(Impl* taskManagerImpl, __code next(...));
11    __code next(...);
12    __code next1(...);
13 } TaskManager;
```

3.3 TaskQueue

Gears OS における TaskQueue は Synchronized Queue で実現される。Worker TaskQueue は TaskManager を経由して Task を送信するスレッドと、Task を取得する Worker 自身のスレッドで扱われる。Gears OS の TaskQueue はマルチスレッドでの操作を想定しているため、データの一貫性を保証する必要がある。そのため、データの一貫性を並列実行時でも保証するために Compare and Swap(CAS) を利用して Queue の操作を行っている。CAS はデータの比較・置換をアトミックに行う命令である。メモリからデータの読みだし、変更、メモリへのデータの書き出しという一連の処理を CAS を利用することで処理の間に他のスレッドがメモリに変更を加えないことを保証することができる。CAS に失敗した場合は置換を行わず、再びデータの呼び出しから始める。

3.4 Workers

Worker は自身の Queue から Task を取得し、Task の Code Gear を実行し、Output Data Gear の書き出しを行っている。Worker は初期化の際に スレッドを生成する。生成されたスレッドはまず、Context を生成する。Context をスレッド毎に生成することで、メモリ空間をスレッド毎に持てるため他のスレッドを止めることはない。Context の生成後は Queue から Task を取得する。Task は Context で実現されているため Worker 自身の Context と Context である Task を入れ替え、Task を実行する。Task の実行後に Data Gear の書き出しと依存関係の解決を行う。その後、Context を Worker の Context に入れ替え、再び Task を取得する。

第4章　Interface

Interface は Gears OS のモジュール化の仕組みである。Interface は呼び出しの引数になる Data Gear の集合であり、そこで呼び出される Code Gear のエントリである。呼び出される Code Gear の引数となる Data Gear はここで全て定義される。Interface を定義することで複数の実装を持つことができる。図 4.1 は Stack の Interface とその実装を表したものである。

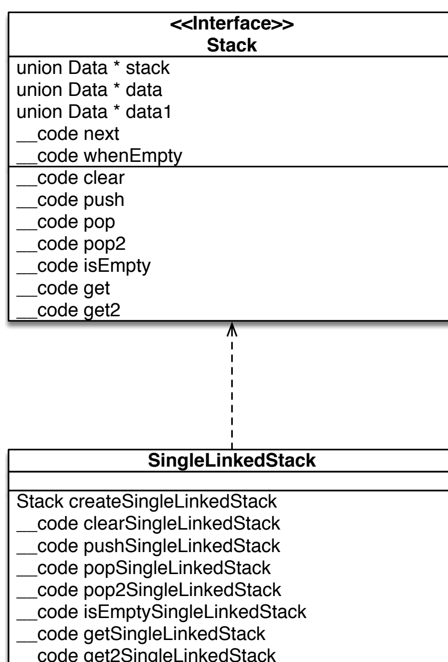

図 4.1: Stack の Interface とその実装

Data Gear は、ノーマルレベルとメタレベルで見え方が異なる。ノーマルレベルの Code Gear では Data Gear の引数に見える。しかし、メタレベルでは Data Gear は Context が持つ構造体である。この見え方の違いを Meta Code Gear である stub Code Gear によって調整する必要がある。

また、CbC は関数呼び出しと異なり、goto による継続で遷移を行う。このため CbC

の継続にはスタックフレームがなく引数を格納する場所がない。

Context は初期化の際に引数格納用の Data Gear の領域を確保する。Code Gear が継続する際にはこの領域に引数の Data Gear を格納する。この領域に確保された Data Gear へのアクセスは Interface の情報から行われる。

ソースコード 4.1 は、pushSingleLinkedStack のソースコードである。ノーマルレベルの Code Gear では Stack の push の操作は、push するデータと次の継続先の Code Gear という引数の集合のように見える。しかしメタレベルでは Context が持つ構造体である。

stub Code Gear では Context が確保した 引数格納用の領域に格納した Data Gear を取り出している。Interface を導入することでノーマルレベルとメタレベルのズレの調整を解決した。

ソースコード 4.1: pushSingleLinkedStack

```
1  __code stackTest1(struct Stack* stack) {
2      Node* node = new Node();
3      node->color = Red;
4      goto stack->push(node, stackTest2);
5  }
6
7  __code pushSingleLinkedStack_stub(struct Context* context) {
8      SingleLinkedStack* stack = (SingleLinkedStack*)context->data[D_Stack
       ]->Stack.stack->Stack.stack;
9      Data* data = context->data[D_Stack]->Stack.data;
10     num Code next = context->data[D_Stack]->Stack.next;
11     goto pushSingleLinkedStack(context, stack, data, next);
12 }
13
14 __code pushSingleLinkedStack(struct SingleLinkedStack* stack, union Data*
        data, __code next(...)) {
15     Element* element = new Element();
16     element->next = stack->top;
17     element->data = data;
18     stack->top = element;
19     goto next(...);
20 }
```

4.1 Interface の定義

ソースコード 4.2 は Stack の Interface である。typedef struct Stack で Interface を定義する。Impl には実装の型が入る。ソースコード 4.2、2〜4 行目の union Data で定義されてるものは、Interface の API で用いる全ての Data Gear である。Interface の全ての API で用いる全ての Data Gear は Interface で定義される。ソースコード 4.2、5〜13 行目の __code で記述されているものは、Interface の API である。ここでは Stack の API である push や pop などの Code Gear となっている。

ソースコード 4.2: Stack の Interface

```
typedef struct Stack<Impl>{
    union Data* stack;
    union Data* data;
    union Data* data1;
    __code whenEmpty(...);
    __code clear(Impl* stack,__code next(...));
    __code push(Impl* stack,union Data* data, __code next(...));
    __code pop(Impl* stack, __code next(union Data*, ...));
    __code pop2(Impl* stack, __code next(union Data*, union Data*, ...));
    __code isEmpty(Impl* stack, __code next(...), __code whenEmpty(...));
    __code get(Impl* stack, __code next(union Data*, ...));
    __code get2(Impl* stack, __code next(union Data*, union Data*, ...));
    __code next(...);
} Stack;
```

通常 Code Gear、Data Gear に参照するためには Context を通す必要があるが、Interface を記述することでデータ構造の API と Data Gear を結びつけることが出来る。これによりノーマルレベルとメタレベルの分離が可能となった。

4.2 Interface の実装の記述

ソースコード 4.3 は Stack の実装の例である。createImpl は実装の初期化を行う関数である。Interface の実装を呼び出す際、この関数を呼び出すことで ソースコード 4.1 4 行目のように実装の操作を呼び出せるようになる。ソースコード 4.3 2 行目は操作する Stack のデータ構造の確保をしている。SingleLinkedStack のデータ構造は ソースコード 4.4 である。ソースコード 4.3 6〜12 行目で実装の Code Gear に代入しているものは Context が持つ enum で定義された Code Gear の番号である。ソースコード 2.2 3 行目で stack->pop へと goto しているが、stack->pop には Code Gear の番号が入っているため実装した Code Gear へと継続する。このため、ソースコード 2.2 では 6 行目の popSingleLinkedStack へと継続している。

ソースコード 4.3: SingleLinkedStack の実装

```
Stack* createSingleLinkedStack(struct Context* context) {
    struct Stack* stack = new Stack();
    struct SingleLinkedStack* singleLinkedStack = new SingleLinkedStack()
    ;
    stack->stack = (union Data*)singleLinkedStack;
    singleLinkedStack->top = NULL;
    stack->push = C_pushSingleLinkedStack;
    stack->pop  = C_popSingleLinkedStack;
    stack->pop2  = C_pop2SingleLinkedStack;
    stack->get  = C_getSingleLinkedStack;
    stack->get2  = C_get2SingleLinkedStack;
    stack->isEmpty = C_isEmptySingleLinkedStack;
```

```
12     stack->clear = C_clearSingleLinkedStack;
13     return stack;
14 }
15
16 __code clearSingleLinkedStack(struct SingleLinkedStack* stack,__code next
       (...)) {
17     stack->top = NULL;
18     goto next(...);
19 }
20
21 __code pushSingleLinkedStack(struct SingleLinkedStack* stack,union Data*
      data, __code next(...)) {
22     Element* element = new Element();
23     element->next = stack->top;
24     element->data = data;
25     stack->top = element;
26     goto next(...);
27 }
```

ソースコード 4.4: SingleLinkedStack のデータ構造

```
1     struct SingleLinkedStack {
2         struct Element* top;
3     } SingleLinkedStack;
4
5     struct Element {
6         union Data* data;
7         struct Element* next;
8     } Element;
```

第5章　メタレベルのコードの自動生成

stub Code Gear などの Meta Code Gear は通常ユーザーレベルからは見ることのできない Meta Data Gear である Context を扱うため、ユーザー自身が記述することは望ましくない。stub Code Gear は Code Gear 毎に記述する必要があるためユーザーの記述量が多くなる。また、stub Code Gear でユーザーが Context から Data Gear を参照するためのコードが非常に煩雑である。このため Meta Code Gear は自動生成されるのが望ましい。そこで Meta Gear を自動生成するためのスクリプトを導入した。また、このスクリプトによって Context の参照をユーザーレベルから隠すことができ、ユーザーレベルの Code Gear もシンプルになった。

これらの Meta Code Gear は本来は OS が動的に生成するべきであるが、現在はコンパイル時に perl スクリプトによって静的に生成を行うことで実現している。

5.1　Meta Code Gear の生成

stub Code Gear は Code Gear 間の継続に挟まれる Meta Code Gear である。必要な Data Gear を Context から参照し、継続する Code Gear へと渡すための Meta Code Gear である。Code Gear 毎に記述する必要があり、継続する Code Gear の引数を見て取り出す Data Gear を選択する。stub Code Gear はユーザーが任意に記述することも出来るが、Meta を扱うため自動生成を行いたい。そのため、stub Code Gear を 自動生成する generate stub を Perl スクリプトで作成した。これにより Code Gear の記述量を約半分にすることができる。

ソースコード 5.1 は ユーザーレベルの Code Gear である。この Code Gear を generate stub によって変換、stub Code Gear の生成を行なったコードがソースコード 5.2 である。

ソースコード 5.1: 変換前の Stack pop

```
1 __code popSingleLinkedStack(struct SingleLinkedStack* stack, __code next(
     union Data* data, ...)) {
2     if (stack->top) {
3         data = stack->top->data;
4         stack->top = stack->top->next;
5     } else {
6         data = NULL;
7     }
```

```
8     goto next(data, ...);
9 }
```

ソースコード 5.2: 変換後の Stack pop

```
1
2 __code popSingleLinkedStack_stub(struct Context* context) {
3         SingleLinkedStack* stack = (SingleLinkedStack*)GearImpl(context,
     Stack, stack);
4         enum Code next = Gearef(context, Stack)->next;
5         Data** O_data = &Gearef(context, Stack)->data;
6         goto popSingleLinkedStack(context, stack, next, O_data);
7 }
8
9 __code popSingleLinkedStack(struct Context *context,struct
     SingleLinkedStack* stack, enum Code next,union Data **O_data) {
10        Data* data = *O_data;
11    if (stack->top) {
12        data = stack->top->data;
13        stack->top = stack->top->next;
14    } else {
15        data = NULL;
16    }
17        *O_data = data;
18    goto meta(context, next);
19 }
```

生成された stub Code Gear は、継続先の Code Gear が引数で指定した Input Data Gear、Output Data Gear を Context から参照している。Gearef は Context から Data Gear を参照するためのマクロである。ソースコード 5.3 1 行目が Gearef マクロの定義である。ソースコード 5.4 はマクロを用いなかった場合の pop stack の stub Code Gear である。Context には Allocation 等で生成した Data Gear へのポインタが格納されている。Code Gear が Context にアクセスする際、ポインタを使用してデータを取り出すため煩雑なコードとなる。そこで Code Gear がデータを参照するための Gearef というマクロを定義した。Gearef は Context から Interface の引数格納用の Data Gear を取り出す。Gearef に Context と Interface の型を渡すことでデータの参照が行える。

GearImpl マクロは Interface の型に包まれた Data Gear から実装の Data Gear を取り出すためのマクロである。ソースコード 5.3 2 行目が GearImpl マクロの定義である。実装の Data Gear を取り出す際も、ポインタでの記述が複雑になってしまうため 同様に GearImpl を定義した。GearImpl は Context と Interface の型、実装の Data Gear 名を指定することで参照する。

ソースコード 5.3: Gearef、GearImpl

```
1 #define Gearef(context, t) (&(context)->data[D_##t]->t)
2 #define GearImpl(context, intf, name) (Gearef(context, intf)->name->intf.
     name)
```

ソースコード 5.4: マクロを用いない stub Code Gear

```
1 __code popSingleLinkedStack_stub(struct Context* context) {
2     SingleLinkedStack* stack = (SingleLinkedStack*)context->data[D_Stack
     ]->Stack.stack->Stack.stack;
3     enum Code next = context->data[D_Stack]->Stack.next;
4     Data** O_data = &context->data[D_Stack]->Stack.data;
5     goto popSingleLinkedStack(context, stack, next, O_data);
6 }
```

また、Code Gear は継続の際 meta へと goto する。Context はすべての Code Gear のリストを持っており、継続先の Code Gear へは enum で対応付けられた Code Gear のアドレスのリストを参照して継続を行う。この meta もスクリプトにより自動生成される。

stub Code Gear を生成するために generate_stub は、ソースコード上の Code Gear を全て取得し、引数からその Code Gear に必要な Data Gear を選択する。

generate_stub は stub Code Gear を生成する際、Code Gear の引数と Interface を照らし合わせ、Code Gear が要求する引数の Data Gear を Context から取り出すための Gearef または GearImpl を決定する。

この時既に stub Code Gear が記述されている Code Gear は stub Code Gear が生成されずに無視される。

generate_stub は Code Gear の変換も行う。ソースコード 5.2 では、popSingleLinkedStack の引数の Output Data Gear を見て、Output Data Gear の格納を行うコードが挿入されている。また、継続のコードが goto meta へと変換されている。

5.2 Context の生成

generate_context は Context を生成する Perl スクリプトである。Context は生成する際に Code Gear のリストとアドレスの対応、引数格納用の Data Gear の生成を行う。

このため Gears OS で新たに Code Gear、Data Gear を定義した際には、Code Gear、Data Gear の enum のリストの更新、引数格納用の Data Gear の Allocation を行うコードの記述を行わなければならなかった。

generate_context を用いることでこれらの煩雑な記述を自動生成で行えるようにした。

generate_context は Context の定義が記述されている context.h を読み、Data Gear を取得する。Code Gear の取得は generate_stub で生成されたコードを読み、ソースコード内の __code 型を見て行う。取得した Code Gear、Data Gear の enum の定義は enumCode.h、enumData.h に生成される。

Code Gear、Data Gear の名前とポインタの対応は generate_context によって生成される enum Code、enum Data を指定することで接続を行う。generate context は、この Code Gear、Data Gear の初期化のコードの initContext の生成も行う。(ソースコード 3.3)

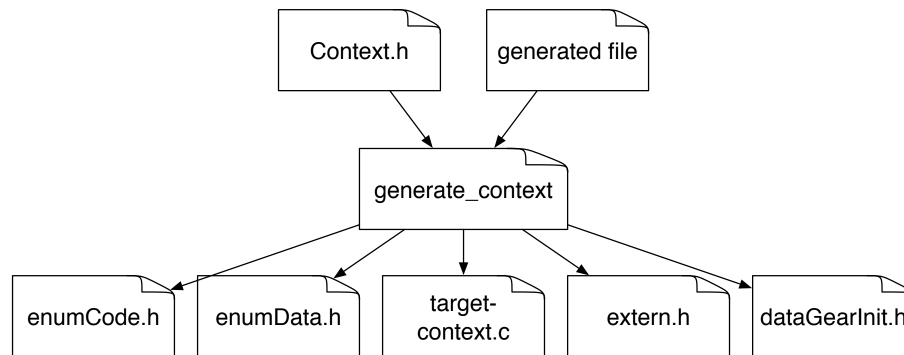

図 5.1: generate_context による Context の生成

また、Gears OS のメタ計算に用いる __code meta や main 関数から継続する start_code、終了処理を行う exit_code も生成される。

これらはまとめて target-context.c として生成される。

ソースコード 5.5: 生成された target-context.c

```
1
2  #include <stdlib.h>
3
4  #include "../context.h"
5
6  void initContext(struct Context* context) {
7      context->heapLimit = sizeof(union Data)*ALLOCATE_SIZE;
8      context->code = (__code(**) (struct Context*)) NEWN(ALLOCATE_SIZE,
       void*);
9      context->data = NEWN(ALLOCATE_SIZE, union Data*);
10     context->heapStart = NEWN(context->heapLimit, char);
11     context->heap = context->heapStart;
12
13     context->code[C_clearSingleLinkedStack]    =
       clearSingleLinkedStack_stub;
14     context->code[C_exit_code]    = exit_code_stub;
15     context->code[C_getSingleLinkedStack]    = getSingleLinkedStack_stub;
16     context->code[C_isEmptySingleLinkedStack]    =
       isEmptySingleLinkedStack_stub;
17     context->code[C_popSingleLinkedStack]    = popSingleLinkedStack_stub;
18     context->code[C_pushSingleLinkedStack]    =
       pushSingleLinkedStack_stub;
19     context->code[C_stack_test1]    = stack_test1_stub;
20     context->code[C_stack_test2]    = stack_test2_stub;
21     context->code[C_stack_test3]    = stack_test3_stub;
22     context->code[C_stack_test4]    = stack_test4_stub;
23     context->code[C_start_code]    = start_code_stub;
24
25 #include "dataGearInit.c"
26
```

```
27 }
28 
29 __code meta(struct Context* context, enum Code next) {
30     // printf("meta %d\n",next);
31     goto (context->code[next])(context);
32 }
33 
34 __code start_code(struct Context* context) {
35     goto meta(context, context->next);
36 }
37 
38 __code start_code_stub(struct Context* context) {
39     goto start_code(context);
40 }
41 
42 __code exit_code(struct Context* context) {
43     free(context->code);
44     free(context->data);
45     free(context->heapStart);
46     goto exit(0);
47 }
48 
49 __code exit_code_stub(struct Context* context) {
50     goto exit_code(context);
51 }
52 
53 // end context_c
```

Context には Allocation 等で生成した Data Gear へのポインタが格納されている。Code Gear は Context を通して Data Gear へアクセスする。Data Gear の Allocation を行うコードは dataGearInit.c に生成される。

第6章　xv6 の CbC への書き換え

xv6 は 2006 年に MIT のオペレーティングシステムコースで教育用の目的として開発されたオペレーティングシステムである。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。

CbC は 継続を中心とした言語であるため状態遷移モデルに落とし込むことができる。xv6 を CbC で書き換えることにより、OS の機能の保証が可能となる。

また、ハードウェア上でのメタレベルの計算や並列実行を行いたい。このため、xv6 を ARM[13] プロセッサを搭載したシングルボードコンピュータである Raspberry pi 用に移植した xv6-rpi を用いて実装する。

6.1　xv6 のモジュール化

xv6 全体を CbC で書き換えるためには Interface を用いてモジュール化する必要がある。xv6 をモジュール化し、CbC で書き換えることができれば、Gears OS の機能を置き換えることもできる。図 6.1 は xv6 の構成図となる。

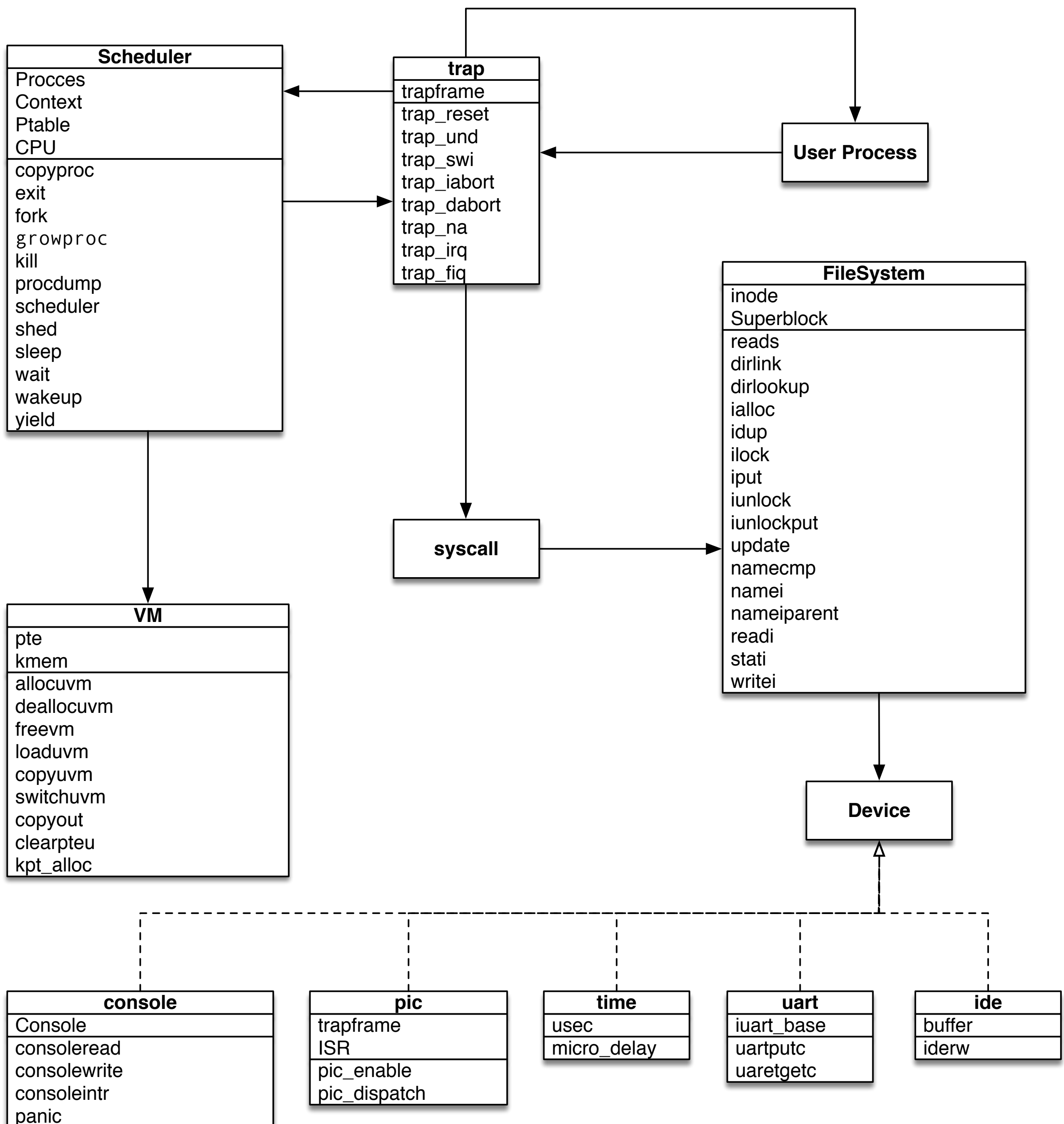

図 6.1: モジュール化した xv6 の構成

6.2 xv6 の構成要素

xv6 はカーネルと呼ばれる形式をとっている。カーネルは OS にとって中核となるプログラムである。xv6 ではカーネルとユーザープログラムは分離されており、カーネルはプログラムにプロセス管理、メモリ管理、I/O やファイルの管理などのサービスを提供する。ユーザープログラムがカーネルのサービスを呼び出す場合、システムコールを用いてユーザー空間からカーネル空間へ入りサービスを実行する。カーネルは CPU のハードウェア保護機構を使用して、ユーザー空間で実行されているプロセスが自身のメモリのみアクセスできるように保護している。ユーザープログラムがシステムコールを呼び出すと、ハードウェアが特権レベルを上げ、カーネルのプログラムが実行される。この特権レベルを持つプロセッサの状態をカーネルモード、特権のない状態をユーザーモードという。

6.2.1 システムコール

ユーザープログラムがカーネルの提供するサービスを呼び出す際にはシステムコールを用いる。ユーザープログラムがシステムコールを呼び出すと、トラップが発生する。トラップが発生すると、ユーザープログラムは中断され、カーネルに切り替わり処理を行う。ソースコード 6.1 は xv6 のシステムコールのリストである。

ソースコード 6.1: xv6 のシステムコールのリスト

```
static int (*syscalls[])(void) = {
        [SYS_fork]    =sys_fork,
        [SYS_exit]    =sys_exit,
        [SYS_wait]    =sys_wait,
        [SYS_pipe]    =sys_pipe,
        [SYS_read]    =sys_read,
        [SYS_kill]    =sys_kill,
        [SYS_exec]    =sys_exec,
        [SYS_fstat]   =sys_fstat,
        [SYS_chdir]   =sys_chdir,
        [SYS_dup]     =sys_dup,
        [SYS_getpid]  =sys_getpid,
        [SYS_sbrk]    =sys_sbrk,
        [SYS_sleep]   =sys_sleep,
        [SYS_uptime]  =sys_uptime,
        [SYS_open]    =sys_open,
        [SYS_write]   =sys_write,
        [SYS_mknod]   =sys_mknod,
        [SYS_unlink]  =sys_unlink,
        [SYS_link]    =sys_link,
        [SYS_mkdir]   =sys_mkdir,
        [SYS_close]   =sys_close,
};
```

6.2.2 プロセス

プロセスとは、カーネルが実行するプログラムの単位である。xv6 のプロセスは、ユーザー空間メモリとカーネル用のプロセスの状態を持つ空間で構成されている。プロセスは独立しており、他のプロセスからメモリを破壊されたりすることはない。また、独立していることでカーネルそのものを破壊することもない。各プロセスの状態は struct proc によって管理されている。プロセスは fork システムコールによって新たに生成される。fork は新しく、親プロセスと呼ばれる呼び出し側と同じメモリ内容の、子プロセスと呼ばれるプロセスを生成する。fork システムコールは、親プロセスであれば子プロセスの ID、子プロセスであれば 0 を返す。親プロセスと子プロセスは最初は同じ内容を持っているが、それぞれ異なるメモリ、レジスタで実行されているため、片方のメモリ内容を変更してももう片方に影響はない。exit システムコールはプロセスの停止を行い、メモリを解放する。wait システムコールは終了した子プロセスの ID を返す。子プロセスが終了するまで待つ。exec システムコールは呼び出し元のプロセスのメモリをファイルシステムのファイルのメモリイメージと置き換え実行する。ファイルには命令、データなどの配置が指定されたフォーマット通りになっていなければならない。xv6 は ELF と呼ばれるフォーマットを扱う。

6.2.3 ファイルディスクリプタ

ファイルディスクリプタは、カーネルが管理するプロセスが読み書きを行うオブジェクトを表す整数値である。プロセスは、ファイル、ディレクトリ、デバイスを開く、または既存のディスクリプタを複製することによって、ファイルディスクリプタを取得する。xv6 はプロセス毎にファイルディスクリプタのテーブルを持っている。ファイルディスクリプタは普通、0 が標準入力、1 が標準出力、2 がエラー出力として使われる。ファイルディスクリプタのテーブルのエントリを変更することで入出力先を変更することができる。1 の標準出力を close し、ファイルを open することでプログラムはファイルに出力することになる。ファイルディスクリプタはファイルがどのように接続するか隠すことでファイルへの入出力を容易にしている。

6.2.4 ファイルシステム

xv6 のファイルシステムはバイト配列であるデータファイルとデータファイルおよび他のディレクトリの参照を含むディレクトリを提供する。ディレクトリは root と呼ばれる特別なディレクトリから始まるツリーを形成している。絶対パスである ”/dir1/dir2/file1” というパスは root ディレクトリ内の dir1 という名前のディレクトリ内の dir2 という名

前のディレクトリ内の file というデータファイルを指す。相対パスである ”dir2/file2” のようなパスは、現在のディレクトリ内の dir2 という名前のディレクトリ内の file というデータファイルを指す。

6.3 xv6-rpi の CbC 対応

オリジナルの xv6 は x86 アーキテクチャで実装されたものだが、xv6-rpi は Raspberry Pi 用に実装されたものである。

xv6-rpi を CbC で書き換えるために、GCC 上で実装した CbC コンパイラを ARM 向けに build し xv6-rpi をコンパイルした。これにより、 xv6-rpi を CbC で書き換えることができるようになった。

6.4 Raspberry Pi

Raspberry Pi は ARM[13] プロセッサを搭載したシングルボードコンピュータである。教育用のコンピューターとして開発されたもので、低価格であり、小型であるため使い勝手が良い。ストレージにハードディスクや SSD を使用するのではなく、SD カードを用いる。HDMI 出力や USB ポートも備えており、開発に最適である。

Raspberry Pi には Raspberry Pi 3、Raspberry Pi 2、Raspberry Pi 1、Raspberry Pi Zero といったバージョンが存在する。

6.5 CbC によるシステムコールの書き換え

CbC によるシステムコールの書き換えは、従来の xv6 のシステムコールの形から、大きく崩すことなく可能である。CbC は Code Gear 間の遷移は goto による継続で行われるため、状態遷移ベースでのプログラムに適している。xv6 を CbC で書き換えることにより、OS のプログラムを状態遷移モデルに落とし込むことができる。これにより状態遷移系のモデル検査が可能となる。図 7.1 は CbC 書き換えによる read システムコールの遷移図である。

ソースコード 6.2 は syscall() におけるシステムコールの呼び出しを行うコードである。システムコールはソースコード 6.1 の関数のリストの番号から呼び出される。CbC でも同様に num で指定された番号の cbccodes のリストの Code Gear へ goto する。ソースコード 6.2 6 行目でトラップフレームからシステムコールの番号を取得する。通常のシステムコールであればソースコード 6.2 13 行目からの分岐へ入るが、cbc システムコールで

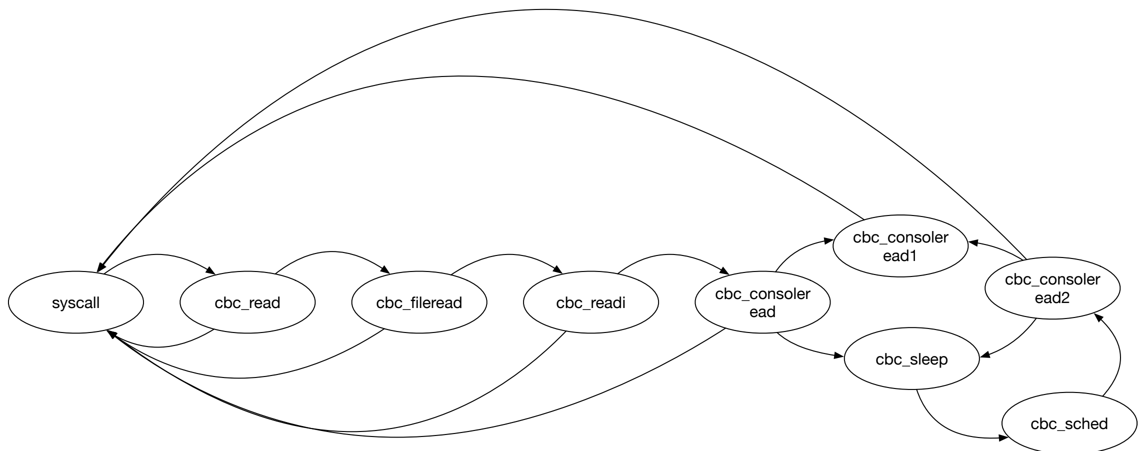

図 6.2: read システムコールの遷移図

あれば ソースコード 6.2 8 行目へ入り、goto によって遷移する。引数に持つ cbc_ret は継続した先でトラップに戻ってくるための Code Gear である。

ソースコード 6.2: syscall()

```
1  void syscall(void)
2  {
3      int num;
4      int ret;
5  
6      num = proc->tf->r0;
7  
8      if((num >= NELEM(syscalls)) && (num <= NELEM(cbccodes)) && cbccodes[
       num]) {
9          proc->cbc_arg.cbc_console_arg.num = num;
10         goto (cbccodes[num])(cbc_ret);
11     }
12 
13     if((num > 0) && (num <= NELEM(syscalls)) && syscalls[num]) {
14         ret = syscalls[num]();
15 
16         // in ARM, parameters to main (argc, argv) are passed in r0 and
       r1
17         // do not set the return value if it is SYS_exec (the user
       program
18         // anyway does not expect us to return anything).
19         if (num != SYS_exec) {
20             proc->tf->r0 = ret;
21         }
22     }  else {
23         cprintf("%d %s: unknown sys call %d\n", proc->pid, proc->name,
       num);
24         proc->tf->r0 = -1;
```

```
25      }
26  }
```

ソースコード 6.3 は、read システムコールであるソースコード 6.4 を CbC で書き換えたコードである。CbC は C の関数を呼び出すことも出来るため、書き換えたい部分だけを書き換えることができる。Code Gear であるため、ソースコード 6.3 7、9 行目のように関数呼び出しではなく goto による継続となる。

ソースコード 6.3: cbc_read システムコール

```
1  __code cbc_read(__code (*next)(int ret)){
2      struct file *f;
3      int n;
4      char *p;
5
6      if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
       {
7          goto next(-1);
8      }
9      goto cbc_fileread(f, p, n, next);
10 }
```

ソースコード 6.4: read システムコール

```
1  int sys_read(void)
2  {
3      struct file *f;
4      int n;
5      char *p;
6
7      if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
       {
8          return -1;
9      }
10
11     return fileread(f, p, n);
12 }
```

ソースコード 6.5 は、ソースコード 6.6 を CbC で書き換えたコードである。継続で Code Gear 間を遷移するため関数呼び出しとは違い元の関数には戻ってこない。このため、書き換えの際には ソースコード 6.5 のように分割する必要がある。プロセスの状態を持つ struct proc は大域変数であるため Code Gear を用いるために必要なパラメーターを cbc_arg として持たせることにした。継続を行う際には必要なパラメーターをここに格納する。

ソースコード 6.5: fileread の CbC 書き換えの例

```
1  __code cbc_fileread1 (int r)
2  {
3      struct file *f = proc->cbc_arg.cbc_console_arg.f;
```

```
    __code (*next)(int ret) = cbc_ret;
    if (r > 0)
        f->off += r;
    iunlock(f->ip);
    goto next(r);
}

__code cbc_fileread (struct file *f, char *addr, int n, __code (*next)(
    int ret))
{
    if (f->readable == 0) {
        goto next(-1);
    }

    if (f->type == FD_PIPE) {
        goto cbc_piperead(f->pipe, addr, n, next);
        goto next(-1);
    }

    if (f->type == FD_INODE) {
        ilock(f->ip);
        proc->cbc_arg.cbc_console_arg.f = f;
        goto cbc_readi(f->ip, addr, f->off, n, cbc_fileread1);
    }

    goto cbc_panic("fileread");
}
```

ソースコード 6.6: 書き換え前の fileread

```
int fileread (struct file *f, char *addr, int n)
{
    int r;

    if (f->readable == 0) {
        return -1;
    }

    if (f->type == FD_PIPE) {
        return piperead(f->pipe, addr, n);
    }

    if (f->type == FD_INODE) {
        ilock(f->ip);

        if ((r = readi(f->ip, addr, f->off, n)) > 0) {
            f->off += r;
        }

        iunlock(f->ip);

        return r;
    }
```

```
24
25      panic("fileread");
26  }
```

ソースコード 6.7 は、ソースコード 6.8 を書き換えたコードである。CbC では cbc_devsw を定義しておりソースコード 6.7 11 行目で次の Code Gear へと継続する。cbc_devsw はソースコード 6.9 で初期化されており、cbc_consoleread へと継続する。

ソースコード 6.7: readi の CbC 書き換えの例

```
1  __code cbc_readi (struct inode *ip, char *dst, uint off, uint n, __code
       (*next)(int ret))
2  {
3      uint tot, m;
4      struct buf *bp;
5
6      if (ip->type == T_DEV) {
7          if (ip->major < 0 || ip->major >= NDEV || !cbc_devsw[ip->major].
       read) {
8              goto next(-1);
9          }
10
11         goto cbc_devsw[ip->major].read(ip, dst, n, next);
12     }
13
14     if (off > ip->size || off + n < off) {
15         goto next(-1);
16     }
17
18     if (off + n > ip->size) {
19         n = ip->size - off;
20     }
21
22     for (tot = 0; tot < n; tot += m, off += m, dst += m) {
23         bp = bread(ip->dev, bmap(ip, off / BSIZE));
24         m = min(n - tot, BSIZE - off%BSIZE);
25         memmove(dst, bp->data + off % BSIZE, m);
26         brelse(bp);
27     }
28
29     goto next(n);
30 }
```

ソースコード 6.8: 書き換え前の readi

```
1  int readi (struct inode *ip, char *dst, uint off, uint n)
2  {
3      uint tot, m;
4      struct buf *bp;
5
6      if (ip->type == T_DEV) {
7          if (ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
       {
```

```
            return -1;
        }

        return devsw[ip->major].read(ip, dst, n);
    }

    if (off > ip->size || off + n < off) {
        return -1;
    }

    if (off + n > ip->size) {
        n = ip->size - off;
    }

    for (tot = 0; tot < n; tot += m, off += m, dst += m) {
        bp = bread(ip->dev, bmap(ip, off / BSIZE));
        m = min(n - tot, BSIZE - off%BSIZE);
        memmove(dst, bp->data + off % BSIZE, m);
        brelse(bp);
    }

    return n;
}
```

ソースコード 6.9: consoleinit

```
void consoleinit (void)
{
    initlock(&cons.lock, "console");
    initlock(&input.lock, "input");

    devsw[CONSOLE].write = consolewrite;
    devsw[CONSOLE].read = consoleread;
    cbc_devsw[CONSOLE].write = cbc_consolewrite;
    cbc_devsw[CONSOLE].read = cbc_consoleread;

    cons.locking = 1;
}
```

ソースコード 6.10 の cbc_consoleread は、ソースコード 6.11 を書き換えたものである。ソースコード 6.10 11、50、行目 では sleep へ継続する際、戻るべき継続先を一緒に送ることで、sleep から consoleread に戻ってくることができる。

ソースコード 6.10: consoleread の CbC 書き換えの例

```
__code cbc_consoleread2 ()
{
    struct inode *ip = proc->cbc_arg.cbc_console_arg.ip;
    __code(*next)(int ret) = proc->cbc_arg.cbc_console_arg.next;
    if (input.r == input.w) {
        if (proc->killed) {
            release(&input.lock);
```

```
8             ilock(ip);
9             goto next(-1);
10        }
11        goto cbc_sleep(&input.r, &input.lock, cbc_consoleread2);
12    }
13    goto cbc_consoleread1();
14 }
15
16 __code cbc_consoleread1 ()
17 {
18     int cont = 1;
19     int n = proc->cbc_arg.cbc_console_arg.n;
20     int target = proc->cbc_arg.cbc_console_arg.target;
21     char* dst = proc->cbc_arg.cbc_console_arg.dst;
22     struct inode *ip = proc->cbc_arg.cbc_console_arg.ip;
23     __code(*next)(int ret) = proc->cbc_arg.cbc_console_arg.next;
24
25     int c = input.buf[input.r++ % INPUT_BUF];
26
27     if (c == C('D')) {  // EOF
28         if (n < target) {
29             // Save ^D for next time, to make sure
30             // caller gets a 0-byte result.
31             input.r--;
32         }
33         cont = 0;
34     }
35
36     *dst++ = c;
37     --n;
38
39     if (c == '\n') {
40         cont = 0;
41     }
42
43     if (cont == 1) {
44         if (n > 0) {
45             proc->cbc_arg.cbc_console_arg.n = n;
46             proc->cbc_arg.cbc_console_arg.target = target;
47             proc->cbc_arg.cbc_console_arg.dst = dst;
48             proc->cbc_arg.cbc_console_arg.ip = ip;
49             proc->cbc_arg.cbc_console_arg.next = next;
50             goto cbc_sleep(&input.r, &input.lock, cbc_consoleread2);
51         }
52     }
53
54     release(&input.lock);
55     ilock(ip);
56
57     goto next(target - n);
58 }
59
60 __code cbc_consoleread (struct inode *ip, char *dst, int n, __code(*next)
       (int ret))
```

```
61 {
62     uint target;
63 
64     iunlock(ip);
65 
66     target = n;
67     acquire(&input.lock);
68 
69     if (n > 0) {
70         proc->cbc_arg.cbc_console_arg.n = n;
71         proc->cbc_arg.cbc_console_arg.target = target;
72         proc->cbc_arg.cbc_console_arg.dst = dst;
73         proc->cbc_arg.cbc_console_arg.ip = ip;
74         proc->cbc_arg.cbc_console_arg.next = next;
75         goto cbc_consoleread2();
76     }
77     goto cbc_consoleread1();
78 }
```

ソースコード 6.11: 書き換え前の consoleread

```
1  int consoleread (struct inode *ip, char *dst, int n)
2  {
3      uint target;
4      int c;
5 
6      iunlock(ip);
7 
8      target = n;
9      acquire(&input.lock);
10 
11     while (n > 0) {
12         while (input.r == input.w) {
13             if (proc->killed) {
14                 release(&input.lock);
15                 ilock(ip);
16                 return -1;
17             }
18 
19             sleep(&input.r, &input.lock);
20         }
21 
22         c = input.buf[input.r++ % INPUT_BUF];
23 
24         if (c == C('D')) {  // EOF
25             if (n < target) {
26                 // Save ^D for next time, to make sure
27                 // caller gets a 0-byte result.
28                 input.r--;
29             }
30 
31             break;
32         }
33 
```

```
34            *dst++ = c;
35            --n;
36
37            if (c == '\n') {
38                break;
39            }
40        }
41
42        release(&input.lock);
43        ilock(ip);
44
45        return target - n;
46    }
```

CbC による書き換えにより、状態遷移ベースへと書き換えることができた。現在はシステムコールの書き換えのみであるが、カーネル全体を書き換えることで、カーネルを状態遷移モデルに落とし込むことができる。

第7章　評価

本研究では、Gears OS のモジュール化、メタレベルの計算の自動生成、xv6 の CbC 書き換えの考察と、システムコールの書き換えを行なった。
　これらの実装についての評価を行う。

7.1　Gears OS のモジュール化

　Gears OS の モジュール化について評価を行う。モジュール化されていない以前の Gears OS ではソースコード 7.1 4 行目のように、Code Gear 直接指定しなければならなかった。Interface を用いた ソースコード 7.2 ではここを stack-¿push のように抽象化することができる。
　また、Gears OS では、ある Data Gear を Code Gear が扱う場合、Code Gear に対応する Data Gear を Context が持つ Data Gear のリストから取り出す必要があるが、ソースコード 7.1 12、13 行目のように Context に引数格納用として型ごとに確保された Data Gear を、複数の Code Gear で使いまわしてしまう問題があった。これも Interface を用いることで、Interface 用に実装で用いる Data Gear を持てるようになり、ソースコード 7.2 12、13 行目のように使うことができるようになった。

ソースコード 7.1: Interface を用いない Gears

```
1  __code cg1 (struct Context* context, struct Element* element) {
2      struct Node* node1 = new Node();
3      element->data = (union Data*)node1;
4      goto meta(context, pushSingleLinkedStack)
5  }
6
7  __code pushSingleLinkedStack(struct Context* context, struct
      SingleLinkedStack* stack, struct Element *element) {
8      ...
9  }
10
11 __code pushSingleLinkedStack_stub(struct Context* context){
12     SingleLinkedStack* stack = &context->data[SingleLinkedStack]->
      SingleLinkedStack;
13     Element *element = &context->data[Element]->element;
14     goto pushSingleLinkedStack(context, stack, data);
```

```
15 }
```

ソースコード 7.2: Interface を用いた Gears

```
1  __code cg1 (struct Context* context, struct Stack* stack) {
2      struct Node* node1 = new Node();
3      goto stack->push((union Data*)node1,cg2);
4  }
5
6  __code pushSingleLinkedStack(struct Context* context, struct
     SingleLinkedStack* stack, union Data* data, __code next(...)) {
7      ...
8  }
9
10 __code pushSingleLinkedStack_stub(struct Context* context){
11     SingleLinkedStack* stack = (SingleLinkedStack*)context->data[D_Stack
     ]->Stack.stack->Stack.stack;
12     Data* data = &context->data[D_Stack]->Stack->data;
13     enum Code next = &context->data[D_Stack]->Stack->next;
14     goto pushSingleLinkedStack(context, stack, data, next);
15 }
```

7.2 Code Gear の変換と Meta Gear の自動生成

CbC はノーマルレベルとメタレベルという異なる階層を持つ言語である。メタレベルのコードの変換と生成は本来コンパイラで行うべきものだが、Code Gear の変換や Meta Gear の生成の考察や評価のため、現在は Perl スクリプトにより、コンパイル時に生成、変換を行なっている。

Code Gear の引数は、ノーマルレベルと、メタレベルによって意味が異なる。ノーマルレベルでは引数はそのまま次の Code Gear へと渡す引数の集合だが、メタレベルでは Context から参照して取り出す Data Gear を表す。この意味のズレを調整するのが stub Code Gear である。

stub Code Gear は、Code Gear 間の継続の間に挟まれる。この stub Code Gear へ継続するための ノーマルレベルの Code Gear の変換作業を行うことは、ノーマルレベルとメタレベルでのズレを調整するために必要な作業である。

また、stub Code Gear は Code Gear 毎に挿入されるため、これが自動生成されることによってソースコードの記述量が大幅に減少した。

7.3 xv6 の CbC 書き換えのための環境構築

xv6-rpi の CbC 書き換えを行うために、ARM 用のクロスコンパイラを build する必要があった。しかし、LLVM 上で実装した CbC では上手くいかなかったため、GCC 上で

実装した CbC を用いることにした。GCC 上で実装した CbC は更新が止まっていたため、現在の GCC へのアップデート作業を行なった。

これにより、GCC 上で実装された CbC が、最新の環境で動作するようになり、ARM 用のクロスコンパイラの build も可能となった。

7.4　xv6 の CbC 書き換え

CbC は Code Gear 間の遷移は goto による継続で行われるため、状態遷移ベースでのプログラムに適している。xv6 を CbC で書き換えることにより、実行可能な OS のプログラムを状態遷移モデルに落とし込むことができる。これにより状態遷移系のモデル検査が可能となる。

xv6 のシステムコールを CbC によって書き換えることで、元の C のソースコードからの大きな変更をすることなしに、図 7.1 のように状態遷移ベースの実行可能なプログラムへと落とし込むことができた。

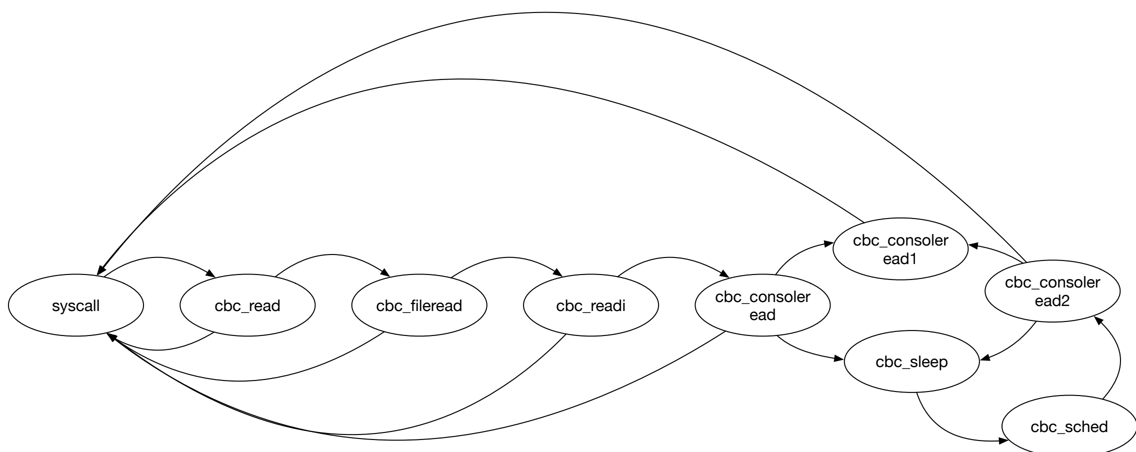

図 7.1: read システムコールの遷移図

また、CbC は C 言語との互換があるため、システムコールのみ CbC へと書き換えるといったことも可能であるため、信頼性を保証したい機能のみを CbC で記述することもできる。

さらに、CbC は Agda に変換できるように設計されているため CbC で記述された実行可能なプログラムが、そのまま Agda による定理証明が可能となる。

これらのため、CbC で記述された実行可能な OS そのものがモデル検査と定理証明が可能となり、OS の信頼性が保証できる。

第8章　結論

本論文では Gears OS のプロトタイプの設計と実装、メタ計算である Context と stub の生成を行う Perl スクリプトの記述を行った。さらに Raspberry Pi 上での Gears OS 実装の考察、xv6 の機能の一部を CbC で書き換えを行った。

Code Gear 、Data Gear を処理とデータの単位として用いて Gears OS を設計した。Code Gear、Data Gear にはメタ計算を記述するための Meta Code Gear、Meta Data Gear が存在する。メタ計算を Meta Code Gear、によって行うことでメタ計算を階層化して行うことができる。Code Gear は関数より細かく分割されてるためメタ計算を柔軟に記述できる。Gears OS は Code Gear と Input/Output Data Gear の組を Task とし、並列実行を行う。

Code Gear と Data Gear は Interface と呼ばれるまとまりとして記述される。Interface は使用される Data Gear の定義と、それに対する操作を行う Code Gear の集合である。Interface は複数の実装をもち、Meta Data Gear として定義される。従来の関数呼び出しでは引数をスタック上に構成し、関数の実装アドレスを Call するが、Gears OS では引数は Context 上に用意された Interface の Data Gear に格納され、操作に対応する Code Gear に goto する。

Context は使用する Code Gear、Data Gear をすべて格納している Meta Data Gear である。通常の計算から Context を直接扱うことはセキュリティ上好ましくない。このため Context から必要なデータを取り出して Code Gear に接続する Meta Code Gear である stub Code Gear を定義した。stub Code Gear は Code Gear 毎に記述され、Code Gear 間の遷移の前に挿入される。

これらのメタ計算の記述は煩雑であるため Perl スクリプトによる自動生成を行なった。これにより Gears OS のコードの煩雑さは改善され、ユーザーレベルではメタを意識する必要がなくなった。

ハードウェア上での Gears OS の実装を実現させるために Raspberry Pi 上での実装を考察した。比較的シンプルな OS である xv6 を CbC に書き換えることにした。

xv6 を CbC で書き換えることによって、実行可能な CbC プログラムで記述された OS がそのまま、状態遷移モデルによるモデル検査、Agda による定理証明が可能となる。

今後の課題は、現在は xv6 のシステムコールの一部のみの書き換えと、設計のみしか行っていないので、カーネル全ての書き換えと、Gears OS の TaskManager の置き換え

を行い、Gears OS の機能を xv6 に組み込む必要がある。また、xv6-rpi は QEMU のみの動作でしか確認してないため、実機上での動作を行う必要がある。

謝辞

本研究の遂行、本論文の作成にあたり、御多忙にも関わらず終始懇切なる御指導と御教授を賜わりました河野真治准教授に心より感謝致します。そして、 共に研究を行い暖かな気遣いと励ましをもって支えてくれた並列信頼研究室の全てのメンバーに感謝致します。最後に、 有意義な時間を共に過ごした理工学研究科情報工学専攻の学友、並びに物心両面で支えてくれた家族に深く感謝致します。

2019 年 3 月
宮城光希

参考文献

[1] Herbert Bos Andrew S.Tanenbaum. Modern Operating Systems. 2015.

[2] Kaito TOKKMORI and Shinji KONO. Implementing continuation based language in llvm and clang. *LOLA 2015*, July 2015.

[3] 河野真治, 伊波立樹, 東恩納琢偉. Code gear、data gear に基づく os のプロトタイプ. 情報処理学会システムソフトウェアとオペレーティング・システム研究会 (OS), May 2016.

[4] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, Vol. 93, No. 1, pp. 55–92, July 1991.

[5] Jean Yang and Chris Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pp. 99–110, New York, NY, USA, 2010. ACM.

[6] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pp. 207–220, New York, NY, USA, 2009. ACM.

[7] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pp. 1–16, Berkeley, CA, USA, 2016. USENIX Association.

[8] Ulf Norell. Dependently typed programming in agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, pp. 1–2, New York, NY, USA, 2009. ACM.

[9] Hokama MASATAKA and Shinji KONO. Gearsos の hoare logic をベースにした検証手法. ソフトウェアサイエンス研究会, Jan 2019.

[10] Russ Cox, Frans Kaashoek, Robert Morris. xv6 a simple, unix-like teaching operating system. https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf.

[11] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[12] GNU Compiler Collection (GCC) Internals. *http://gcc.gnu.org/onlinedocs/gccint/.*

[13] ARM Architecture Reference Manual. *http://infocenter.arm.com/help/topic/com.arm.doc.subset.architecture.reference/index.html.*

[14] Raspberry Pi — Teach, Learn, and Make with Raspberry Pi. https://www.raspberrypi.org.

[15] Zhiyi Wang. xv6-rpi. https://code.google.com/archive/p/xv6-rpi/, 2013.