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

GearsOS の分散ファイルシステム設計

2022 年 3 月
March 2022

一木 貴裕
Ikki Takahiro

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 山田 孝治
Supervisor: Prof. Yamada Koji

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)　　山田 孝治

印

(副　査)　　岡﨑 威生

印

(副　査)　　玉城 絵美

印

(副　査)　　河野　真治

要旨

当研究室ではOSの信頼性の検証を目的としたOSであるGearsOSを開発している。GearsOSはユーザレベルとメタレベルを分離して記述を行うことで拡張性と信頼性の補償を目指している。GearsOSはContinuation based C(以下CbC)で記述されており,Gearというプログラミング概念を持つ。GearsOSは現在開発途上であるため、現在は言語フレームワークとして実装されている。OSとして起動するためにこれから実装が必要な機能が多く存在しており、その中の一つとして分散ファイルシステムが挙げられる。GearsOSの分散ファイルシステムでは、当研究室が開発している分散フレームワークChristieの仕組みを用いる。ChrisitieはGearsOSと異なるGearというプログラミング概念をもち、DataGearManagerというDataPoolをノード間でsocket接続することで通信を実装している。GearsOSのファイルはDataGearManagerと同様な構成として、ファイルであると同時に通信処理そのものに相当する。本研究ではChristieのDataGearManagerをファイルとして用い、その通信のAPIの構成を行った。

GearsOSのファイルシステムは従来のファイルシステムの問題点を解決した実装を取り込めるような構成にしたい。従来のファイルシステムの問題点の一部として、ファイルシステムはデータベースでなくレコード単位での操作が行えない、バックアップや証明書などの機能はアプリケーションに依存している点などが挙げられる。将来的な実装を鑑みてファイルシステムの実装を行う。

また、GearsOSを純粋に利用して記述を行うプロジェクトは本研究が初めてとなる。分散ファイルシステムの構成を行うと同時に、GearsOSの言語フレームワークとしての評価と検討を行った。

Abstract

In our laboratory, we are developing GearsOS, an operating system for verifying the reliability of operating systems, which aims to compensate for scalability and reliability by separating user-level and meta-level descriptions. GearsOS is written in Continuation based C (CbC) and has the programming concept of Gears. Since GearsOS is still under development, it is currently implemented as a language framework, and there are many functions that need to be implemented in order to run as an OS, one of which is a distributed file system. One of them is a distributed file system. The distributed file system of GearsOS uses the distributed framework Christie, which is developed by our laboratory. Christie has a different programming concept called "Gear" from GearsOS, and implements communication by connecting a DataPool called DataGearManager to a socket between nodes. The files in GearsOS are structured in the same way as DataGearManager, so that they are not only files but also correspond to the communication process itself. In this study, we used Christie's DataGearManager as a file and configured the API for its communication.

The file system of GearsOS should be configured in such a way that it can incorporate implementations that solve the problems of conventional file systems. Some of the problems with conventional file systems are that they are not databases and cannot be manipulated on a record-by-record basis, and that functions such as backup and certificates are dependent on the application. The file system will be implemented in view of future implementations.

In addition, this research is the first project that purely uses GearsOS for description. While constructing the distributed file system, we also evaluated and examined GearsOS as a language framework.

発表履歴

- 一木 貴裕, 河野 真治. 分散フレームワーク Christie による Block chain の実装. 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, 2019
- 一木 貴裕, 河野 真治. GearsOS の分散ファイルシステムの設計. 情報処理会 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, 2021

目 次

図 目 次

表 目 次

第1章　GearsOSにおけるファイルシステム

コンピュータにおけるファイルシステムは必要不可欠な存在であると言える。コンピュータで行われる計算やそれに使われるデータは全てデバイス上にファイルとして保持され、ファイルという概念なしにはユーザーがコンピュータ上で用いられる資源の管理はほぼ不可能であると言える。コンピュータの技術発展と普及に伴いファイルは物理的な存在場所に囚われない、つまりネットワークへの接続を通して別マシン上のどこからでもアクセスが行える存在である必要が生じてきた。物理的な位置を問わずネットワークを通してファイルを共有する機能を有するファイルシステムを特に分散ファイルシステムと呼ぶ。分散ファイルシステムはネットワークに配置されたファイルに対して、ローカルに保存されたファイルとほとんど相違なく取り扱いが行える透過性が求められる。

当研究室では信頼性の保証と拡張性の高さを目指したOSプロジェクトである、GearsOSの開発を行なっている。GearsOSはテストコードでなく、AgdaやCoqなどの型式手法に含まれる定理証明支援系やモデル検査を用い、実施にOSとして動作するコードそのものを検証することで信頼性の補償を目指している。また、GeasOSはノーマルレベルとメタレベルを分離して記述するContinuation based C(CbC)にて記述される。ノーマルレベルとはユーザが行いたい計算をさし、メタレベルとはノーマルレベルの計算を行うための計算をさす。ノーマルレベルのプログラムのメタレベルに対して自由に差し替えを行うことにより、コードの整合性の検証やプログラムのデバッグを行えるような作りとなっている。

現状においてGearsOSは言語フレームワークとして実装されており、実際にOSとして起動するためにはこれから実装が必要となる機能が複数存在している。その中の一つとして分散ファイルシステムが挙げられる。GearsOSのファイルシステムは既存のOSが持つファイルシステムの問題点の解決や従来ではアプリケーションが担っているバックアップを始めとした機能を搭載したファイルシステムの構成を目指して開発を行いたいと考えた。問題点の解決の一部として、既存のファイルシステムとは異なりファイルシステムをデータベースとして構成したい。レコードでファイルデータを取り扱うことによりファイルの更新や操作を簡潔に行い、またFileSystemのAPIを総括してトランザクションとしたい。加えて、データのバックアップについてもOSに搭載したい。従来のように

バックアップデータをユーザが管理するのではなく、OS が管理をすることによりバックアップデータ自体の紛失を防ぎたい。

また、GearsOS の分散ファイルシステムは当研究室が開発している分散フレームワークである Christie の仕組みを用いて構成する。Christie は通信を形成する接続形態 (Topology) の中枢を持たない、自律分散の実現を目的に開発されている。GearsOS のファイルシステムでは、Christie によるファイル通信を構成することで自律分散なファイルシステムの構築を目指したい。

加えて、Christie の持つ Gear 概念による簡潔な分散処理と、処理を行うノードごとの接続 (Topology 形成) を管理する TopologyManager の機能を用いることによって複雑な分散処理の構成を簡潔化が期待できる。

第2章　Continuation based C

Continuation based C(以下 CbC) とは C 言語の拡張言語であり、関数呼び出しではなく軽量継続と呼ばれる処理を主体に記述をおこなう。GearsOS は現状では CbC を拡張した言語フレームワークとして実装されている。CbC における継続とは通常の関数呼び出しと異なり、スタック操作を行わず、次のコードブロックに jmp 命令を用いて移動する処理を指す。コードブロックごとの変数などの環境を保存しないため軽量継続と呼ばれる。また、CbC ではこの軽量継続にて再帰呼び出しを行うことでループ処理を行う。

2.1　Gear 概念

CbC にはCodeGear と DataGear という二つのプログラミング概念が存在する。CodeGear は関数に代わり宣言されるプログラミング単位と呼べる。CbC プログラミングでは連続した CodeGear 間を遷移していくことで構成される。CodeGear は goto という命令で遷移をしていくが、その際に DataGear と呼ばれる任意の型を設定した変数データを次の CodeGear へ引き渡していく。そのため、DataGear は CodeGear が処理の際に参照する専用の変数データであると言える。特に、CodeGear 実行の際に入力される DataGear を InputDataGear、CodeGear の処理後に出力される DataGear を OutputDataGear と呼ぶ。CodeGear は InputDataGear を参照しながら処理を実行し、OutputDataGear として次の CodeGear へ必要なデータを出力する。CodeGear と DataGear の関係を図 2.1 に示す。

CodeGear は通常の関数呼び出しのようにスタックを持たないため、一度 CodeGear から別の CodeGear へ継続した場合、元の処理へ戻ってくることができない。言い換えると、CodeGear から別の CodeGear へ goto で遷移した際、元の CodeGear 内で使用されていた変数データなどの環境は全て破棄される。この点から GearsOS のループ処理は CodeGear の再帰呼び出しで実装することが望ましく、また goto で遷移をした際、goto より後に記述した処理は実行されることなく次の処理へ移るため、これを意識したプログラミングが必要となる。

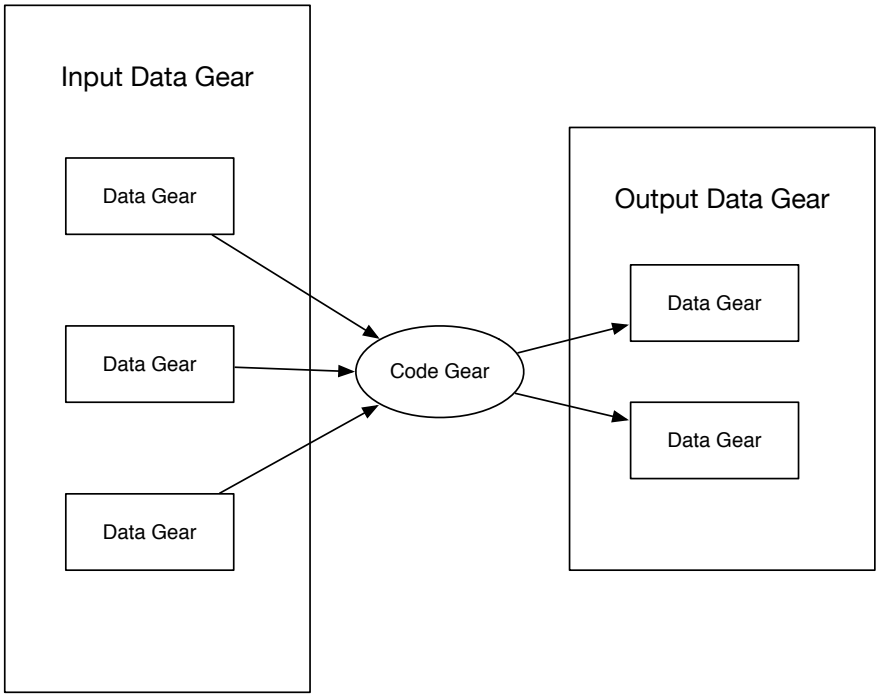

図 2.1: CodeGear と DataGear の関係

2.2 CbC の例題

CbC の簡単なプログラムをソースコード 2.1 に示す。CG1, CG2, はそれぞれ CodeGear であり、__code で宣言されている。CodeGear には関数の引数のように、InputDataGear として任意の型の変数を定義する。CodeGear への遷移をする際は goto コマンドを使用し、遷移先の CodeGear 名と指定された DataGear と同じ型変数を入力する。

今回のソースコードでは、main から二つの int 型の DataGear の入力と共に CG1 へ goto で遷移、続いて CG2 へ CG1 での処理結果を DataGear として引き渡し遷移、最後に exit をするという流れとなる。

ソースコード 2.1: CbC の例題

```
1 __code CG2(int num3){
2   printf("num = %d\n", num3);
3   exit(0);
4 }
5
6 __code CG1(int num, int num1){
7   int num2 = num + num1;
8   goto CG2(num2);
9 }
```

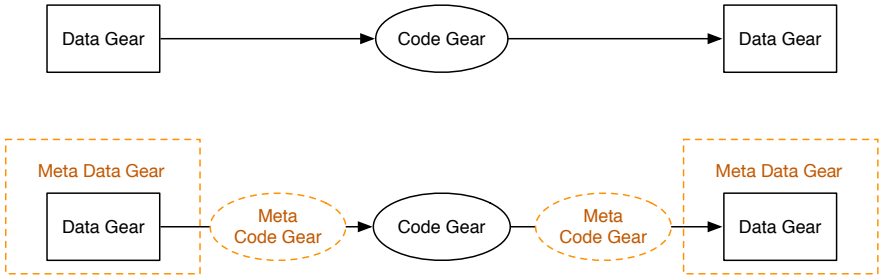

図 2.2: メタレベルを考慮した場合の Gear の関係性

```
10
11 int main(){
12   int a = 2;
13   int b = 3;
14   goto CG1(a, b);
15 }
```

2.3 meta レベルの Gear

図 2.1 で示した CodeGear と DataGear の関係性は単純に連続した複数の CodeGear が DataGear を介してやりとりをしているだけのように見える。しかし、実際には CodeGear の遷移の際には、プログラマが記述するノーマルレベルからは見えないメタレベルの計算が行われている。メタ計算とはノーマルレベルのプログラムを行うための計算を指し、メタレベルを考慮した上で CbC の CodeGear, DataGear の遷移を図式化すると図 2.2 の下段のように表現できる。CbC では CodeGear の遷移に必要な遷移先の CodeGear への DataGear の受け渡しやデータの整合性の確認などは CodeGear ごとに準備された MetaCodeGear と呼ばれるメタな CodeGear で行われ、MetaCodeGear で使用される DataGear を MetaDataGear と呼ぶ。また、CodeGear の実行直前に呼び出される MetaCodeGear を特に StubCodeGear と呼ぶ。

MetaCodeGear と MetaDataGear はプログラマが直接実行するのではなく、ビルド時に実行される、Perl スクリプトによって実装されたトランスコンパイラによって自動的に生成される。しかし、自動生成される StubCodeGear に問題があったり、特殊な処理を組み込みたい場合はプログラマが直接記述することも可能である。トランスコンパイラはすでに実装された StubCodeGear は上書きしないため、この場合、プログラム上に StubCodeGear を直接記述したい CodeGear 名の後ろに `_stub` を付け加えた CodeGear を記述すれば良い。

第3章　GearsOS

GearsOS は当研究室が開発する、OS 自体の信頼性の確保と拡張性の高さを目指した OS である。GearsOS は Continuation based C を用いることにより、ノーマルレベルとメタレベルの分離を行い、プログラマが記述したノーマルレベルのコードをメタレベルから信頼性の検査を行えるような仕組みを作ることにより、OS 自身の信頼性の向上を目指している。メタレベルのプログラムは Perl スクリプトによるトランスコンパイラにて自動生成されるように構成されており、基本的にユーザーはノーマルレベルのプログラムの記述のみ行うことで、メタレベル分離の恩恵を受けられる仕組みとなっている。プログラマは.cbc ファイルに対して記述を行うことでプログラムの構成を行うが、実際に動作するプログラムはトランスコンパイラにより meta 部分の記述が行われた純粋な.c ファイルが動作する。

現状では CbC の言語フレームワークとして実装されており、OS として実際に起動するためには実装が必要となる機能が多く存在している。必要な機能を CbC による記述により実装していくことで OS の完成を目指す。

3.1　Interface

GearsOS の重要な仕様として Interface が存在する。Interface の役割の一つとして、通常のプログラミングにおけるモジュール化の仕組みと同様の役割を持つ。GearsOS に実装されている Queue の Interface をソースコード 3.1 に示す。

ソースコード 3.1: Queue のインターフェース

```
1  typedef struct Queue<>{
2      union Data* queue;
3      union Data* data;
4
5      __code whenEmpty(...);
6      __code clear(Impl* queue, __code next(...));
7      __code put(Impl* queue, union Data* data, __code next(...));
8      __code take(Impl* queue, __code next(union Data* data, ...));
9      __code isEmpty(Impl* queue, __code next(...), __code whenEmpty(...));
10     __code next(...);
11 } Queue;
```

GearsOS における Interface はヘッダファイルに対して記述を行い、API として用いたい CodeGear とその CodeGear に入出力される DataGear を記述する。5 から 10 行目は CodeGear の宣言である。また、2, 3 行目の union Data 型の変数は API の CodeGear で使用される DataGear を記述する。

コード内の CodeGear の第 1 引数は Impl*型の変数となっており、Interface を引き継いだ Implement のポインタとなる。これは CbC はスタックを持たず、複数の CodeGear 間をまたいで必要な情報は DataGear として持ち運ぶ必要があるため、DataGear として Implement のポインタを設定している。また、この Implement のポインタは goto での遷移の際はトランスコンパイラによる補完により、goto の引数として記述する必要はない。

`__code next(...)` はその CodeGear が呼び出される際に入力される別の CodeGear へのポインタであり、その API の CodeGear の処理後の遷移先の CodeGear を指定することができる。`__code whenEmpty(...)` も同様に別 CodeGear へのポインタである。9 行目の CodeGear では next と WhenEmpty と二つの CodeGear のポインタを入力することにより、処理の結果に応じた複数の遷移先を用意するように構成している。

Interface に加え、Implementation は DataGear とみなすことができる。また、GearsOS では後述の Context と呼ばれる環境内に全ての DataGear と CodeGear が保持されている。Context 内では Interface や Implement で定義された API と変数は構造体として記録され、またそれら全ての構造体は共用体 (union) の Data 型に登録されている。そのため、Interface と Implement は DataGear もしくは CodeGear の一時的な保管場所と見なすこともできる。

3.2 Implementation

GearsOS は Interface の実装となる Implement の形定義ファイルが実装されている。Implement の定義には Interface と同様にヘッダファイルを記述する。ソースコード 3.2 に Queue インターフェースの実装の型定義となる SynchronizedQueue.h を示す。Implement は実装元の Interface を必ず決定する必要があるため、1 行目のように impl 名の後に実装元の Interface 名を記述する必要がある。

作成した Implement は実装元の Interface 名の Impl 型として使用することが可能となり、Impl 内で記述した変数は後述のコンストラクタで行われる処理によって、プログラム上で impl*型変数->変数名で参照が行えるようになる。また、GearsOS は全ての Interface, Impl の情報を構造体としてコンパイルする際に Context に記録するため、ソースコード中 2,3,4 行では Impl に別の Interface を構造体として記述することで、プログラム内で他の Interface を実装したプログラム、もしくは Interface を純粋な構造体としてを呼び出すことができる。

ソースコード 3.2: Queue.h の実装となる SynchronizedQueue.h

```
1 typedef struct SynchronizedQueue <> impl Queue {
2   struct Element* top;
3   struct Element* last;
4   struct Atomic* atomic;
5 } SynchronizedQueue;
```

3.3　GearsOS のプログラム例

GearsOS のプログラムの一例として先述したソースコード 3.1、3.2 を Interface,Imlement として継承した SynchronizedQueue.cbc の一部分をソースコード 3.3 に示す。GearsOS には Implement が記述されたヘッダファイルから、cbc ファイルの雛形を自動生成する impl2cbc.pl が導入されている。雛形には 7 行目の Queue*を返す createSychronizedQueue とその中の処理、加えて 22 行目の putSynchronizedQueue のような Interface のヘッダファイル内で記述された CodeGear が用意されている。

プログラム内で呼び出したい Interface は 2,3 行目のような記述が必要となる。7 から 20 行目に記述されている createSynchronizedQueue は Queue インターフェースを SynchronizedQueue で実装する際のコンストラクタである。関数呼び出しで実装されており、返り値は Interface のポインタである。

コンストラクタ内の記述を解説すると、8,9 行目にて Queue と SynchronizedQueue のアロケーションを行い、14 行目の記述にて QueueInterface 内の DataGear である queue へ SynchronizedQueue へのポインタを格納している。8, 10 行目で new 演算子が使われているがこれは GearsOS 独自のシンタックスの一つである。コンパイル時に後述の Context が持つ DataGear のヒープ領域のアロケーションを行うマクロへ置き換えられる。10 から 13 行目の記述では初期は Implement の型定義ファイルに記述された変数の宣言が行われている。当然プログラマは、宣言した変数に任意の状態が設定されるように変更することができる。13 行目では atomic 型の interface を AtomicReference 型で呼び出している。15 から 18 行目は Interface の定義にて記述された API と cbc プログラム内の CodeGear の紐付けを行っている。

ソースコード 3.3: SynchronizedQueue.cbc の記述の一部

```
1 #include "../context.h"
2 #interface "Queue.h"
3 #interface "Atomic.h"
4 #include <stdio.h>
5
6
7 Queue* createSynchronizedQueue(struct Context* context) {
8     struct Queue* queue = new Queue();
9     struct SynchronizedQueue* synchronizedQueue = new SynchronizedQueue()
    ;
```

```
10      synchronizedQueue->top = new Element(); // allocate a free node
11      synchronizedQueue->top->next = NULL;
12      synchronizedQueue->last = synchronizedQueue->top;
13      synchronizedQueue->atomic = createAtomicReference(context);
14      queue->queue = (union Data*)synchronizedQueue;
15      queue->take  = C_takeSynchronizedQueue;
16      queue->put   = C_putSynchronizedQueue;
17      queue->isEmpty = C_isEmptySynchronizedQueue;
18      queue->clear = C_clearSynchronizedQueue;
19      return queue;
20  }
21
22  __code putSynchronizedQueue(struct SynchronizedQueue* queue, union Data*
        data, __code next(...)) {
23      Element* element = new Element();
24      element->data = data;
25      element->next = NULL;
26      Element* last = queue->last;
27      Element* nextElement = last->next;
28      if (last != queue->last) {
29          goto putSynchronizedQueue();
30      }
31      if (nextElement == NULL) {
32          struct Atomic* atomic = queue->atomic;
33          goto atomic->checkAndSet(&last->next, nextElement, element, next
        (...), putSynchronizedQueue);
34      } else {
35          struct Atomic* atomic = queue->atomic;
36          goto atomic->checkAndSet(&queue->last, last, nextElement,
        putSynchronizedQueue, putSynchronizedQueue);
37      }
38  }
```

22 行目から 38 行目は Interface で宣言された putAPI の実装部分である。Interface で宣言された API と対応する CodeGear は.cbc ファイル上では API 名 + Impl 名で宣言する必要がある。コンストラクタ内 13 行目にて atopmiReference の呼び出しを行っているが、atomicInterface へのポインタは最終的に返り値の queue が保持しているため、atomicInterface で定義された API の CodeGear へ遷移することができるようになる。現状の GearsOS では一度、Interface のポインタ型をプログラム内で宣言し、コンストラクタで生成した実装のポインタを代入、そして goto 文を用いて遷移するという冗長な記述が必要となっているため、トランスコンパイラにより改善するといった手段が考えられる。

3.4 Context

Context とは GearsOS における従来の OS のプロセスに相当する概念であるとされる。英単語としての context は意味として前後関係や状況、環境を意味し、軽量継続を主体として処理が行われる GearsOS において、Context は継続に必要なデータ (CodeGear,

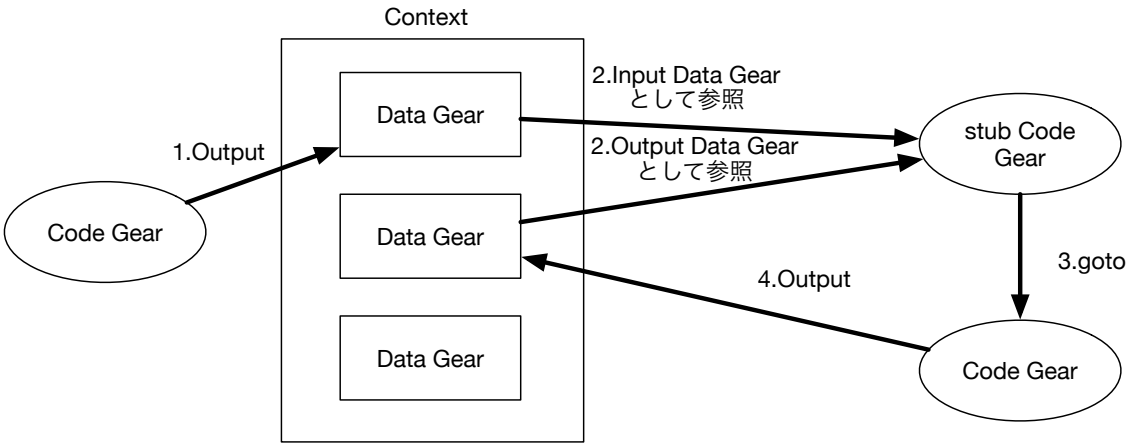

図 3.1: CodeGear の遷移にて行われる Context に対する DataGear 参照

DataGear) を全て保持している。GearsOS では CodeGear の遷移の際にこの Context に対して DataGear の一時的な書き込みや、遷移先のポインタなどの情報を書き込み、Context を持ち運ぶことにより処理の整合性を保っている。ノーマルレベルのプログラム上では意識することは少ないが、メタレベルでは全ての CodeGear は必ず Context を DataGear として呼び出す仕組みとなっている。そのため Context は Gear 概念で言うと MetaDataGear に相当する。

図 3.1 に CodeGear が遷移する際の Context に対する DataGear の書き込みまたは呼び出しを示す。CodeGear が遷移する際に OutputDataGear を Context に対して書き込みを行う。続いて遷移先の stubCodeGear が Context から遷移先 CodeGear が必要とする InputDataGer と出力する必要がある OutputDataGear を参照する。stubCodeGear で DataGear の準備が完了したあと、本体の CodeGear へ遷移し処理により出力されたデータを OutPutDataGear として書き出す。以上の手順で軽量継続の際に Context の参照が行われる。

Context はノーマルレベルから直接参照を行わない。Context をユーザーが任意の操作することはノーマルレベルとメタレベルの分離した意味が無くなってしまうためである。Context に対する DataGear の操作は stubCodeGear のみならず CodeGear 内でも行われるが、それらの記述はトランスコンパイラの自動生成により行われる。先述の new 演算子がその一例として挙げられる。もし new 演算子でなく、直に Context の操作が行えてしまうと、並行に動作している他の User Context の中身を書き換えるなどプログラム処理に支障が出るような不正な改ざんが行えてしまう。

Context はプロセスに相当するため、ユーザープログラムごとに Context が存在する。この Context を User Context と呼ぶ。また実行されている GPU や CPU ごとにも Context

が必要となり、これを CPU Context と呼ぶ。加えて OS として User Context や CPU Context を含めた、全体を管理するための Context も必要となる。これは Kernel Context や KContext と呼ばれている。これらの Context は特に、GearsOS に実装されている par goto と呼ばれる並列処理用の構文の内部構成などのメタレベルのプログラミングの際に強く意識する必要がある。GearsOS では KernelContext が複数の User Context と CPU Context を管理し、ノーマルレベルのプログラムに応じて Worker に対して Task を割り振ることで分散処理の機構が実現されている。

3.5　トランスコンパイル後の実行プログラム

先にも述べたように、GearsOS 上で記述するプログラムは拡張子が.cbc となる cbc ファイルへ記述を行う。しかし、実際の動作ファイルは cbc を Perl スクリプトによりトランスコンパイルにより stubCodeGear などのメタレベルの処理やマクロの書き換えがされた、拡張子.c ファイルとなる。ソースコード 3.4 に SynchronizedQueue.cbc(ソースコード 3.3) をトランスコンパイルした結果出力された SyncheonizedQueue.c の一部である。

cbc からの c 言語へのトランスコンパイルはノーマルレベルと分離したいメタレベルの記述が追加、もしくは置き換えが行われる。GearsOS プログラミングではユーザープログラマは出力後の.c ファイルについて、つまりメタレベルに対して意識する必要がなくなるのが理想である。しかし、開発者側はトランスコンパイラの新たなシンタックスの導入やデバッグ、メタレベルの処理の置き換えなどメタ部分の処理の知識を持つべきである。また、現状のトランスコンパイラは意図しないバグを持っている可能性があるため、通常のユーザプログラムのデバッグなどの際にも.c ファイルを確認する機会がある。

ソースコード 3.4 中の 53 行目から 58 行目までの putSynchronizedQueue_stub は、cbc ファイルに記述された putSynchronizedQueue の stubCodeGear である。stubCodeGear は元の CodeGear 名 + _stub で宣言される。stubCodeGear は inputDataGear として Context を要求する。54 行目では GearImpl マクロを用いて Interface の実装型のポインタを Context から呼び出す。55, 56 行目に記述されている Gearef マクロは context の Interface 内で宣言されている DataGear を参照するマクロである。55 行目の場合、QueueInterface 内で宣言されている DataGear である data を呼び出している。56 行目では共用体として取り扱われている nextCodeGear を呼び出す。CodeGear 本体に必要な inputDataGear を impl で呼び出してから本体の CodeGear へ遷移する。

本体の CodeGear では、ノーマルレベルでの操作を禁止したい Context に対する操作が追記される。goto による遷移の直前では、遷移先の CodeGear に対して出力する DataGear を Context に対して書き戻すための処理が行われる。34 行目から 39 行目などに見られるように、Context の DataGear の保存場所に対する書き込みの際にも Gearef マクロが使われる。Gearef コマンドは Gearef(Context 名、Interface 名)->DataGear 名で指定される。

また、GearsOS 独自の new 演算子は Context の DataGear の保存場所に対してメモリ確保を行うマクロである ALLOCATE に書き換えられる。

ソースコード 3.4: SynchronizedQueue.c

```
1  #include "../context.h"
2  // include "Queue.h"
3  // include "Atomic.h"
4
5  #include <stdio.h>
6
7  Queue* createSynchronizedQueue(struct Context* context) {
8      struct Queue* queue = &ALLOCATE(context, Queue)->Queue;
9      struct SynchronizedQueue* synchronizedQueue = &ALLOCATE(context,
       SynchronizedQueue)->SynchronizedQueue;
10     synchronizedQueue->top = &ALLOCATE(context, Element)->Element; //
       allocate a free node
11     synchronizedQueue->top->next = NULL;
12     synchronizedQueue->last = synchronizedQueue->top;
13     synchronizedQueue->atomic = createAtomicReference(context); // not
       used
14     queue->queue = (union Data*)synchronizedQueue;
15     queue->take  = C_takeSynchronizedQueue;
16     queue->put   = C_putSynchronizedQueue;
17     queue->isEmpty = C_isEmptySynchronizedQueue;
18     queue->clear = C_clearSynchronizedQueue;
19     return queue;
20 }
21
22
23 __code putSynchronizedQueue(struct Context *context,struct
       SynchronizedQueue* queue, union Data* data, enum Code next) {
24     Element* element = &ALLOCATE(context, Element)->Element;
25     element->data = data;
26     element->next = NULL;
27     Element* last = queue->last;
28     Element* nextElement = last->next;
29     if (last != queue->last) {
30         goto meta(context, C_putSynchronizedQueue);
31     }
32     if (nextElement == NULL) {
33         struct Atomic* atomic = queue->atomic;
34         Gearef(context, Atomic)->atomic = (union Data*) atomic;
35         Gearef(context, Atomic)->ptr = (union Data**) &last->next;
36         Gearef(context, Atomic)->oldData = (union Data*) nextElement;
37         Gearef(context, Atomic)->newData = (union Data*) element;
38         Gearef(context, Atomic)->next = next;
39         Gearef(context, Atomic)->fail = C_putSynchronizedQueue;
40         goto meta(context, atomic->checkAndSet);
41     } else {
42         struct Atomic* atomic = queue->atomic;
43         Gearef(context, Atomic)->atomic = (union Data*) atomic;
44         Gearef(context, Atomic)->ptr = (union Data**) &queue->last;
45         Gearef(context, Atomic)->oldData = (union Data*) last;
```

```
46          Gearef(context, Atomic)->newData = (union Data*) nextElement;
47          Gearef(context, Atomic)->next = C_putSynchronizedQueue;
48          Gearef(context, Atomic)->fail = C_putSynchronizedQueue;
49          goto meta(context, atomic->checkAndSet);
50      }
51  }
52
53  __code putSynchronizedQueue_stub(struct Context* context) {
54      SynchronizedQueue* queue = (SynchronizedQueue*)GearImpl(context,
        Queue, queue);
55      Data* data = Gearef(context, Queue)->data;
56      enum Code next = Gearef(context, Queue)->next;
57      goto putSynchronizedQueue(context, queue, data, next);
58  }
```

3.6 SynchronizedQueue

GearsOS には先行研究 [1] にて Gears シンタックスを用いて実装された Queue が存在している。Queue は単純な Queue としての機能を実装した SingleLinkedQueue と、マルチスレッドによる複数のプロセスからのアクセスにも対応するための SynchronizedQueue が存在する。

SynchronizedQueue は複数のプロセスからのアクセスが発生した際のデータの一貫性の保証方法として、CAS(Check and Set, Compare and Swap) を採用している。CAS は値の比較、更新をアトミックに行う命令である。CAS は更新前と更新後の値を受け取り、更新前の値と書き換え先のメモリ番地の実際の値と比較し、同じだった場合、データ競合がないため、データの更新を行う。値が異なった場合は他のプロセスから書き込みが行われたとみなされ、値の更新に失敗する。

GearsOS では CAS を行うための Interface を Atomic として定義している。Atomic の Interface をソースコード 3.5 に示す。ptr はデータ格納先のポインタのポインタ、oldData,newData はそれぞれ更新前の値と更新後の値を示す。__code next には CAS が成功した場合の遷移先を、__code fail には失敗した場合の遷移先の CodeGear を指定する。

SynchronizedQueue ではデータの put もしくは take の際、要素の先頭もしくは最後尾の要素のアドレスに対して CAS が行われる (ソースコード 3.3)

ソースコード 3.5: Atomic.h

```
1  typedef struct Atomic<>{
2      union Data* atomic;
3      union Data** ptr;
4      union Data* oldData;
5      union Data* newData;
6      __code checkAndSet(Impl* atomic, union Data** ptr, union Data*
       oldData, union Data* newData, __code next(...), __code fail(...));
```

```
7     __code next(...);
8     __code fail(...);
9 } Atomic;
```

第4章　分散フレームワークChristie

分散フレームワーク Christie は当研究室が開発する Java 言語で記述された分散フレームワークである。Christie は特定のサーバーを中心として分散処理 Topology を形成するのではなく、各ノード同志が平等な立場として動作する自律分散を目的に開発されている。自律的に接続が増加していく Topology に対して、特定のプロトコルによる通信構成は技術分散が発生し全体の統一性が損なわれてしまう。そのため、プロトコルでなく特定の key に対してデータを書き込みを行うことでデータベースアクセス的に通信を構成する仕組みとなる。

Christie は将来的に GearsOS に組み込むことが考えられており、CbC と似てはいるが異なった別の Gear という概念が存在している。

また、分散処理を行う複数のノード接続を任意の形の Topology に自動で形成してくれる機能である、TopologyManagear が存在する。Christie ではノード同士の接続を煩雑な記述を行うことなく、TopologyManager に一任することで簡潔な記述で分散処理が行える。

4.1　gear 概念

Christie の gear は四種類存在し、以下のように名付けられている。

- CodeGear(以下 CG)
- DataGear(以下 DG)
- CodeGearManager(以下 CGM)
- DataGearManager(以下 DGM)

CodeGear はクラスやスレッドに相当する。DataGear は変数データであり、CodeGear 内でアノテーションを用いて記述する。また、Christie の DataGear は key とデータがペアで管理されている。この二点は GearsOS と同様の名称であり、CodeGear を DataGear を介して CodeGear への遷移を繰り返す形が Christie の仕組みの根幹となる点が GearsOS と類似点であると言える。

CodeGearManager はいわゆるノードに相当している。CodeGearManager は CodeGear を管理し、CodeGear に必要な DataGear が揃った場合にその CodeGear を Worker で実行

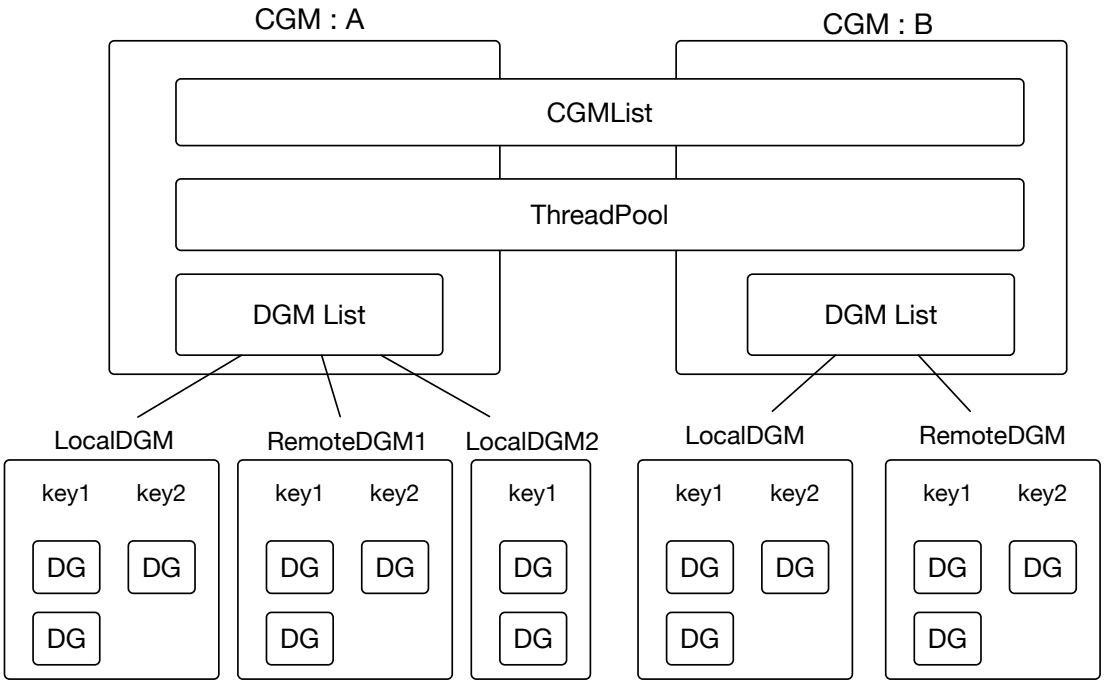

図 4.1: Christie のそれぞれの Gear 関係図

する。また、CodeGearManager はそれぞれ独立した DataGearManager を所持している。DataGearManager はデータプールに相当しており、put 操作された全ての DataGear が key とデータのペアで保存されている。また、DataGearManager は LocalDataGearManager と RemoteDataGearManager に区分される。図 4.1 に Christie の Gear の関係図を示す。

4.2　LocalDGM と RemoteDGM

図 DataGearManager は LocalDGM と RemoteDGM の二種類が存在する。4.2 に DataGearManager による通信の関係性を示す。LocalDGM は所有している CodeGearManager 自身に対応する DGM である。LocalDGM に put 操作を行うことで自身の持つ key に対して DataGear を送ることができる。

RemoteDGM は別の CodeGearManager が持つ LocalDGM の proxy に相当する。つまり、RemoteDGM は別の CodeGearManager に必ず対応しており、RemoteDGM に put 操作すると対応した CGM が持つ LocalDGM へ DataGear を put することができる。図 4.1 に示す。

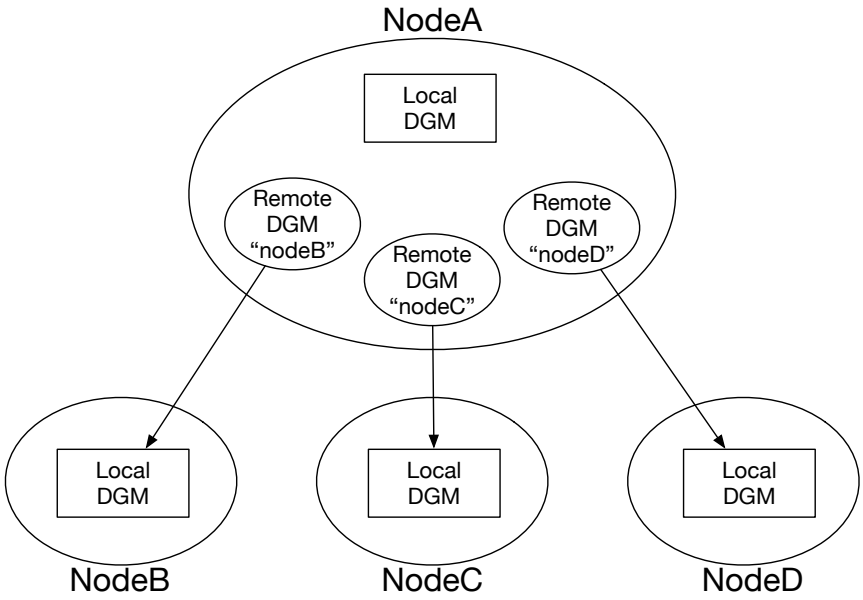

図 4.2: RemoteDGM を介したノード接続

Christie のノードどうしのやりとりはこの RemoteDGM を介して行われる。つまり RemoteDGM を作成することはノードどうしの接続を行うことにあたる。また、RemoteDGM は単方向の通信にあたるため、複数のノードを双方向で通信させたい場合は双方に相手の RemoteDGM を作る必要がある。

4.3　DataGear アノテーション

CodeGear には処理に必要な DataGear を記述する必要がある。DataGear の取り出し操作は以下の四つに分けられる。

Take 先頭の DG を読み込み、その DG を削除する、DG が複数ある場合、この動作を用いる。

Peek 先頭の DG を読み込むが、DG が削除されない、そのため、特に操作をしない場合は同じデータを参照し続ける。

TakeFrom(Remote DGM name) RemoteDGM name を指定することで、その接続先 (Remote) の DGM から Take 操作を行える。

PeekFrom(Remote DGM name) RemoteDGM name を指定することで、その接続先 (Remote) の DGM から Peek 操作を行える。

DataGear の呼び出しは Take 操作が基本となる。Peek 操作は特定の DG を参照し続けたい場合に用いるが、Peek アノテーションを単独で用いる場合、処理次第では無限ループが生じてしまう場合があるため気をつける必要がある。DataGear は CodeGearManger が持つ Local な DataGearManager に対して参照を行う。

4.4　Christie のプログラミング例

ソースコード 4.14.2 に Christie の簡単なプログラム例を示す。

StartCodeGear を継承したソースコード 4.1、StartHelloWorld は CbC や C 言語における main 部分に相当する。main 部分では CodeGearManager をポート番号指定することで作成を行う。また、CGM に対して処理させたい CodeGear を setup メソッドを用いることで、CGM に対して CG を持たせることができる。RemoteDGM の作成などを行う。

CodeGear を継承したソースコード 4.2、OddCodeGear は CodeGear となる。CodeGear は CGM に setup され、また必要な DataGear が全て揃うことで初めて CGM に実行される。CodeGear 内部で次に処理したい CodeGear を CGM に setup することもできる。

ソースコード 4.1: StartOddEven

```
1  package christie.example.OddEven;
2
3  import christie.codegear.CodeGearManager;
4  import christie.codegear.StartCodeGear;
5
6  public class StartOddEven extends StartCodeGear {
7
8      public StartOddEven(CodeGearManager cgm) {
9          super(cgm);
10     }
11
12     public static void main(String[] args){
13         int finishCount = 10;
14         CodeGearManager odd = createCGM(10001);
15         CodeGearManager even = createCGM(10002);
16         odd.setup(new OddCodeGear());
17         even.setup(new EvenCodeGear());
18         odd.createRemoteDGM("even","localhost",10002);
19         even.createRemoteDGM("odd","localhost", 10001);
20         odd.getLocalDGM().put("odd", 1);
21         odd.getLocalDGM().put("finishCount",finishCount);
```

```
22          even.getLocalDGM().put("finishCount", finishCount);
23
24      }
25  }
```

ソースコード 4.2: OddCodeGear

```
1   package christie.example.OddEven;
2
3   import christie.annotation.Peek;
4   import christie.annotation.Take;
5   import christie.codegear.CodeGear;
6   import christie.codegear.CodeGearManager;
7
8   public class OddCodeGear extends CodeGear {
9       Take
10      int odd;
11
12      Peek
13      int finishCount;
14
15      Override
16      protected void run(CodeGearManager cgm) {
17          System.out.println(odd + " : odd");
18          if (finishCount + 1 != odd) {
19          getDGM("even").put("even", odd + 1);
20          cgm.setup(new OddCodeGear());
21          }
22      }
23  }
```

4.5 TopologyManager

Christie にはノード接続の Topology を自動形成するための機能である TopologyManager が備わっている。通常の分散処理プログラムの記述の際にはノードどうしの接続形成は煩雑な記述が必要となってしまう。Christie ではこの TopologyManager に通信接続を一任することにより、気軽に通信形成が行えるようになる。正確には Topology に参加を表明したノードに対して名前を与え、任意の Topology の形に従って各ノードに RemoteDGM を作成させる。

Chrisitie に存在している TopologyManager の形成方法として静的 Topoplogy と動的 Topology がある。静的 Topology はプログラマが任意の形の Topoplogy とノードの配線を dot ファイルに記述し、TopologyManager に参照させることで自由な形の Topology を形成する。現時点では静的 Topology での Topology 形成は dot ファイルに記述した参加ノード数に実際に参加するノードの数が達していない場合、動作しないという制約が存在している。

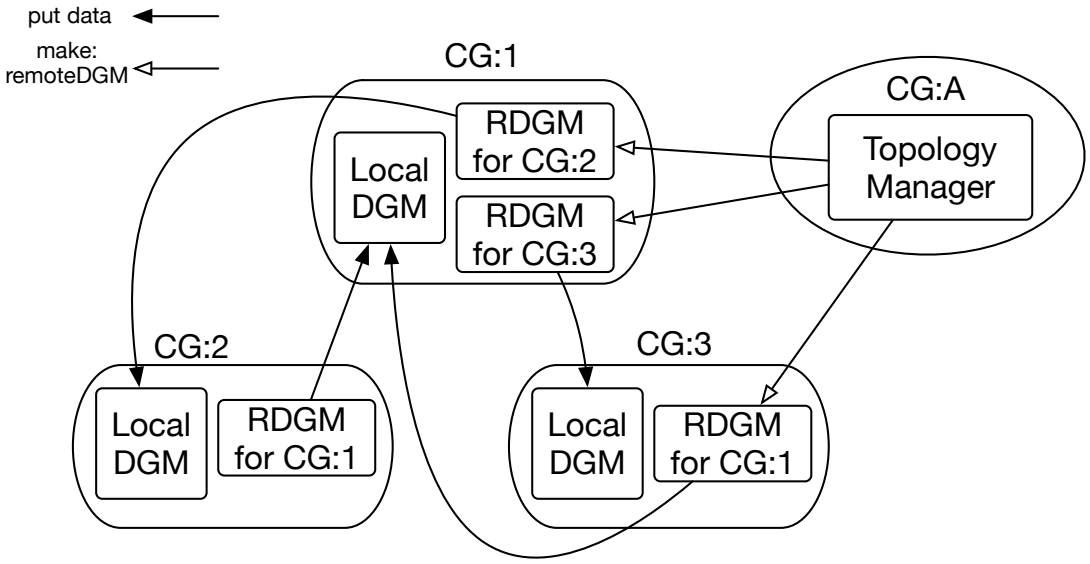

図 4.3: TopologyManager にて Tree 型 Topology を形成する際の図

動的 Topology は TopologyManager 内にすでに設計された Topology の形状を選択した上で参加を表明したノードに対して自動的にノードの配線を行う。例えば Tree を構成する場合、参加したノードから順番に root から近い役割を与えていく。また、TopologyManager は CodeGearManager の一つが運用する。図 4.3 に TopplogyManager にて簡単な Tree 型 Topology を形成する際の図を示す。図中の場合、CGA が TopoplogyManager を所持しており、TopologyManager によって CG1 が root、CG2 と CG3 が CG1 の子として役を割り振られ、接続される形となる。

第5章　GearsFileSystemの設計と実装

本章では GearsOS の FileSystem(以下 GearsFS) の実装について解説する。GearsOS のファイルは分散ファイルシステム的に大域的な資源として、複数のプロセスから競合的なアクセスが可能を可能としたい。

GearsFS は分散フレームワーク Christie の仕組みの一部を応用する。ファイルは Christie における DataGearManager 的に実装を行い、ファイルの送受信は、ファイル内データをレコード単位で分割し、DataGear として送受信することで実現する。

Christie は自律分散システムを目指した分散フレームワークである。Christie の仕組みにより、GearsFS は通信の中枢となるサーバーノードが必要なくなるような、自律分散なファイルシステムを目指す。

5.1　GearsFSのファイル構成

GearsOS におけるファイルのデータは任意の型を持ったレコード (構造体) に断続的に分割されて構成され、それを保持するファイルは単なる DataGear の ListQueue として実装される。つまり、GearsFS におけるファイルの読み書きは従来の分散ファイルシステムのようなプロトコルを用いず、データベースアクセス的な処理で構成される。プロトコルを用いず、最低限なデータアクセスによる通信を構成することで、分散通信技術の見通しをよくする狙いがある。また、CodeGear は Transaction と言えるので key の書き込みは Transaction に閉じられ、動作が明確になる。

ファイルの読み込みの際は Queue に対して順次データレコードの Take 操作を繰り返し、連続したレコードからファイルの中身を構築することで実現する。つまり、一見として Christie のファイルは Queue とその要素 (Element) として構成される。データのリストとなる Queue の構造を図 5.1 に示す。

しかし、単純に一つの Queue に対しファイルとして直接読み書きを行うと、複数のノードからのアクセスされた際の整合性や、セキュリティの面で問題が発生する。そのため、GearsFS のファイルは主体となるデータを保持する主要な Queue とは別に input もしくは outputStream となる Queue を持ち合わせている。そのため GearsFS ファイルは複数の役割を持った DataGear を持つリストとして実装される。GearsOS のファイル構造へのアクセスを図 5.2 に示す。ファイル読み込みもしくは書き込みの際にはそれぞれ input もしく

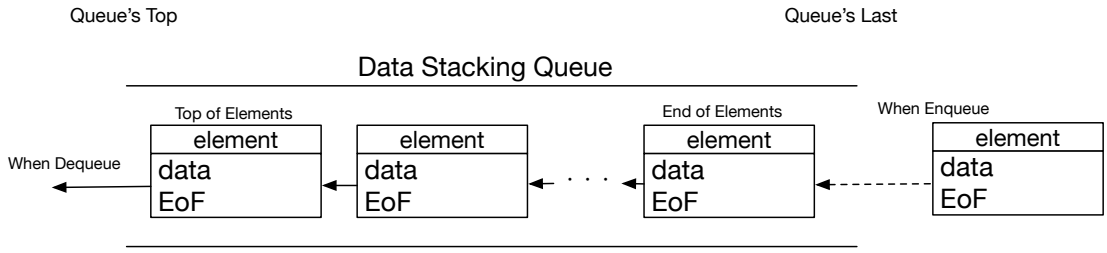

図 5.1: FileData Stacking Queue

は output の際に stream に対してアクセスを行う形でファイルに対する読み書きが行われ、それぞれの Queue はリスト内で固有の keyname を保持している。つまり、ファイルに対する読み書きを行う際は、Queue リストに対して input もしくは output の streamQueue の key に対して put 操作、もしくは take 操作を行えば良い。これは、Chrsitie における DataGear の key とその要素に相当し、それらをリストとして持ち合わせているファイルは DataGearManager に相当する。

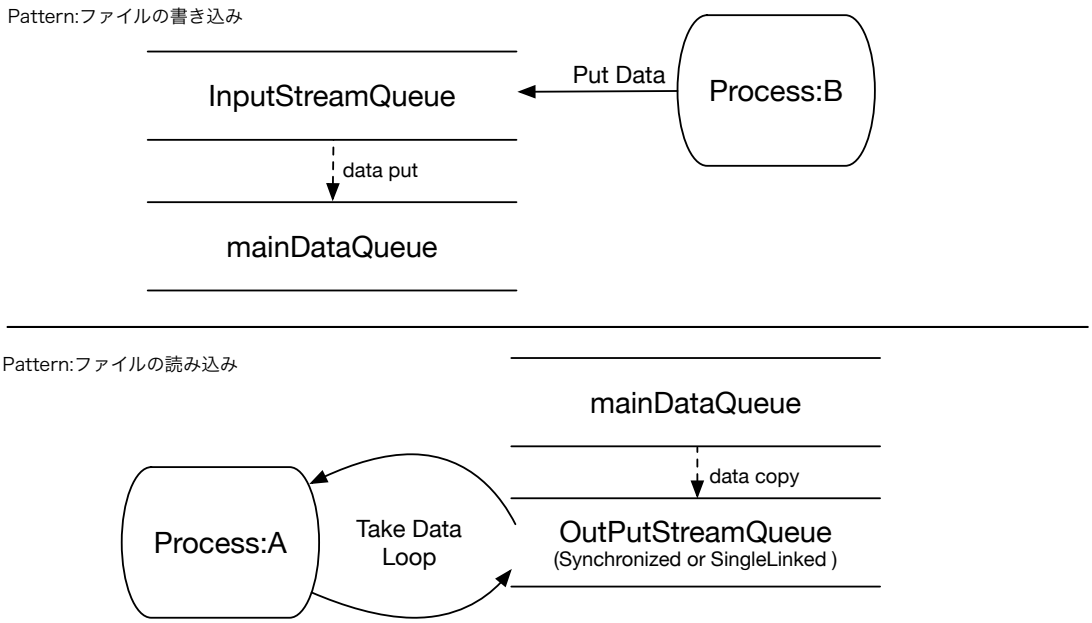

図 5.2: GearsFS の呼び出し/書き込み

ファイルに対して書き込み、つまり従来の write にあたる処理を行う場合は InputQueue の

key を指定して put 操作を行う。最終的に InputQueue に格納されたデータは InputQueue から取り出され、mainQueue に対して格納される。

ファイルの読み込み (既存のファイルシステムでの read) を行う際は OutputQueue に対して Take 操作を行えば良い。OutputQueue には mainQueue の要素が複製されており、OutputQueue 内の要素を全て Take のループにより呼び出す。ファイルを呼び出す側は取り出したレコードを順番に読むことでファイルを構築する。

GearsOS のファイルは大域的な資源として同時に複数のプロセスから参照が可能になる作りにしたい。OutputQueue は複数のプロセスからファイル読み込みが行われた際に、データの整合性が失われてしまう危険性がある。そのため、OutputQueue は CAS(Compare And Swap) を実装した SynchronizedQueue を用いる。しかしファイルのアクセス権限の設定などにより、データアクセスが単一のプロセスからしか許さないもしくは想定されない環境では読み込みの高速化とメモリの軽量化のため、SingleLinked な (単純な)Queue を用いることも考えられる。ファイルに対する書き込みや MainQueue は単一プロセスのみからのアクセスとなるため SingleLicked なキューで実装する。

ファイルのレコードは例題中では単純にファイル内文字列を行ごとに区切り、FileString 構造体に書き込んだものとなる。5.1 に FileString 構造体を示す。str 部分にファイル内文字列を格納し、レコードが全て送信したことを示す EoF フラグが付属している。ファイルレコードは Gears 側の DataGear として利用も行えるように、Context に DataGear として登録された構造体である。

FileString はテストに用いられる単純なレコードである。実用性のあるレコードを考察した場合、ファイルに変更が加えられた場合 Queue に格納するレコードを行順に構成し直す必要性が生じるなど多くの問題が存在する。また、GearsFS ではファイルの型を OS が区別できることを目指しており、レコードの型を最適な形に変更することによって複数のファイルの型に対応したい。例として、テキストデータの場合、git や Mercurial などのバージョン管理システムのように、変更差分をレコードとして Queue に格納し、Queue のレコードを Top から参照していくことで構築を行う。また、画像ファイルなどファイルの中身の変更が行われない形式は、ファイル内のデータをブロック長に区切り扱うといった手段が考えられるなど、ファイルの型に応じてレコードの構造体を任意に構成することができる。

ソースコード 5.1: 例題のレコードとして使われる FileString 構造体

```
1 typedef struct FileString <> {
2   char str[1024];
3   int size;
4   int EoF;
5 } FileString;
```

5.2　WordCount 例題

GearsFS の API の構成を WordCount 例題を通して行った。WordCount 例題とは、指定したファイルの中身を読み取り、文字数と行数列、加えて文字列を出力するという例題である。この例題により連続したレコードを Queue から順次読みだす処理を構築した。コード 5.2 に CbC で記述した、Unix ファイルに対して WordCount を行うプログラムの一部を示す。

ファイルと WordCount の接続は UNIX のシェルのようにプログラムの外で行われる。ファイルは事前に C 言語の FILE 型構造体を用いて開かれ、行列である wc->strTable へ内部文字列を行区切りで保存されている。Main からファイルの Open が完了した後、CodeGear:putStrin に遷移し、strTable から文字列を行ごと呼び出し CountUp へ入力し遷移する。CodeGear:CountUp にて文字列の出力と文字数を読み取り記録、そして putString へ遷移する。ファイルの文字列がある間は putString と CountUp をループし続け、putString は strTable の中身がなくなった (EOF) なら showResult へ遷移する形となる。図 5.3 に CG の遷移図を示す。

WordCount の FileOpen と WordCount 処理を別ノード上で行うことで、ファイルの読み込みとファイルの送信の構成が行える。

ソースコード 5.2: Unix ファイルに対する WordCount.cbc の一部

```
1  __code putString(struct WcImpl* wc, int line, __code next(...)){
2    if (wc->strTable[line] != NULL){
3      wcString* string = NEW(wcString);
4      string->str = wc->strTable[line];
5      goto countUp(string, next);
6    } else {
7      goto showResult(next);
8    }
9  }
10
11 __code countUp(struct WcImpl* wc, wcString* string, __code next(...)) {
12   printf("「countUp%」s\n", string->str);
13   wc->lineNum = wc->lineNum + 1;
14   int num = wc->lineNum;
15   wc->wordNum = wc->wordNum + strlen(string->str);
16   Gearef(context, Wc)->line = num;
17   goto putString(num, next);
18 }
19
20 __code showResult(struct WcImpl* wc, __code next(...)) {
21   printf("Number of Words is 「%」d\n", wc->wordNum);
22   printf("EOF and finish codes\n");
23   goto next(...);
24 }
```

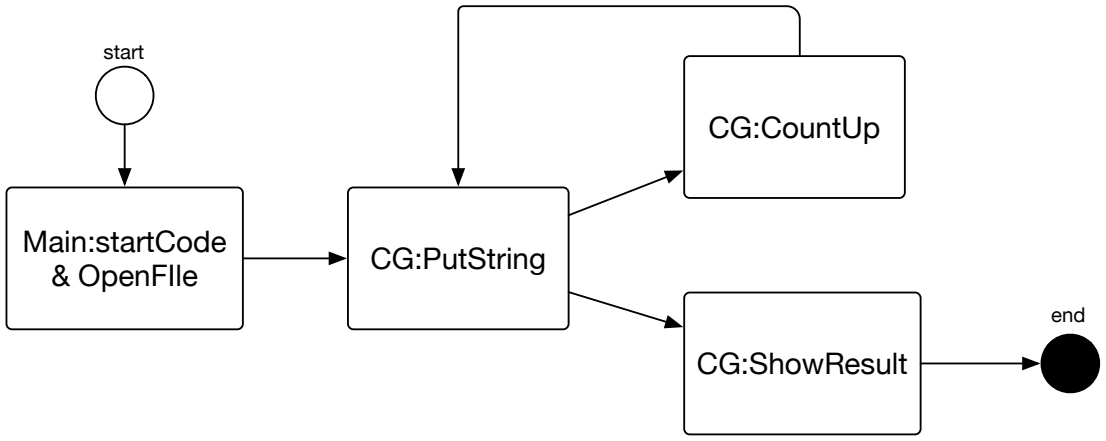

図 5.3: Unix ファイルに対する WordCount の CG 遷移

5.3　GearsFile API による WordCount

GearsFS は Christie の LocalDataGearManager と RemoteDataGearManager による通信の仕組みを用いてファイルデータの送受信を構成する。これを GearsFile API と呼ぶ。ChrstieAPI ではファイルを DataGearManager と見なすことができ、Local 上に Remote 先のファイルの proxy となる RemoteDGM を作成し、これに対し Local のファイルへの書き込みと同様に操作を行うことで自動的に通信を行ってくれる。

Chiristie の DataGearManager の仕組みを基準とした GearsFile API による WordCount の設計を行った。図 5.4 に GearsFile API による WordCount を示す。GearsFile API によるファイルデータ通信は 2 ペアの Local/RemoteDGM を通して行われる。RemoteDGM は接続先のノードが持つ LocalDGM のプロキシであり、RemoteDGM に対して put 操作を行うことで、相手の持つ LocalDGM へのデータの送信が行える。NodeA 側が任意のファイルを開き、ファイル内の行ごとの文字列をデータとして NodeB 側に対応する RemoteDGM に put する。NodeB 側は自身の LocalDGM からデータを得て、WordCount の処理を行ったのちに NodeA に対応する RemoteDGM に対してそのデータに対する処理が完了したことを通知する Ack(acknowledgement) を put する。Ack を受け取った Open 側のノード B は再び続きのファイル内文字列をデータとして送信する。ここまでの処理が繰り返され、ファイル内の文字列が全て処理されたら、Open 側は EoF(End of File) を Count 側に対して通知、NodeB は最後に WordCount の結果を RemoteDGM に対して put し Open 側に送信する。そして、双方の RemoteDGM を閉じることにより通信を終了させる。

図中では RemoteDGM を複数作成することにより通信のやりとりを行っているが、WordCount でない純粋なファイルの送信などの場合、DGM の Socket が持つ Ack のみでも通

信を構成することが可能である。そのためデータの一方的な送受信のみなら DGM のペアは一つだけでも通信が行うことができる。

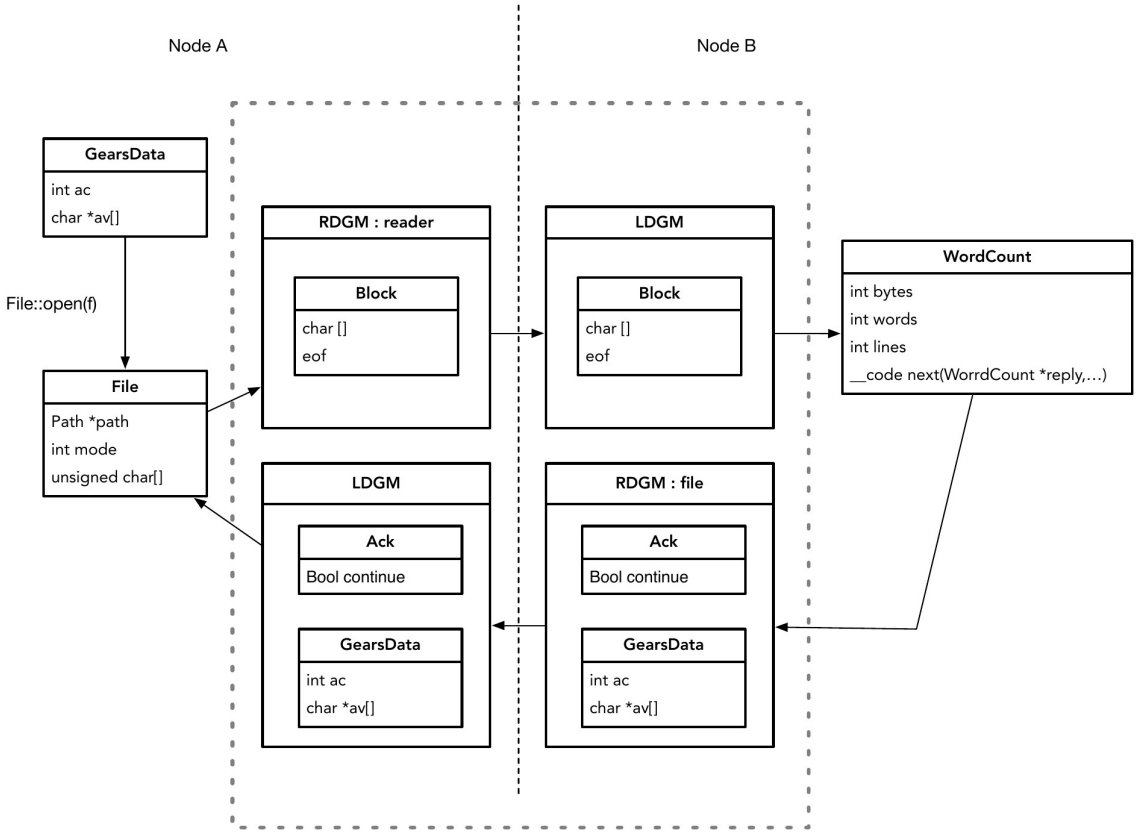

図 5.4: GearsFile API による WordCount

5.4　GearsOS 上の Socket 通信

RemoteDGM と LocalDGM の接続の際には LocalDGM が持つ socket に対して RemoteDGM が接続を行い、Data の書き込み先の key を指定して socket 経由でコマンドとして送信する。

socket を所有した Queue による WordCount 例題の記述を行うことで GearsOS における socket 通信の実装を行った。接続元と接続先の socket は、Queue の作成時に行われ、socket へのアクセスは Queue に新しく API として追記を行った。ソースコード 5.3 に socket 操作を追加した Queue の Interface を示す。ソースコード 5.5 に LocalDGM 側にあたる、Socket 付きの Queue の CodeGear:getData 部分を示す。加えてソースコード 5.6 に RemoteDGM

側にあたる Socket 付きの Queue の CodeGear:sendData 部分を示す。CodeGear:getData では socket などによる Error 発生時や EoF フラグがあるデータを受信した場合の遷移先を指定する必要ある。

ソースコード 5.3: socket の操作を追加した Queue Interface

```
typedef struct CQueue<>{
    union Data* cQueue;
    union Data* data;


    __code whenEmpty(...);
    __code whenEOF(...);
    __code clear(Impl* cQueue, __code next(...));
    __code put(Impl* cQueue, union Data* data, __code next(...));
    __code take(Impl* cQueue, __code next(union Data* data, ...));
    __code isEmpty(Impl* cQueue, __code next(...), __code whenEmpty(...))
   ;
    __code getData(Impl* cQueue, __code next(...), __code whenEOF(...));
    __code next(...);
} CQueue;
```

二つの Queue は main プログラムにより socket の接続が行われ、一方が文字列の送信を行い、もう一方が Count 処理を行うことによって WordCount が行われる。socket は C 言語の Socket インターフェースを用いており、socket の Implement のコンストラクタにあたる createLocalDGMQueue にて呼び出され、Impl のメンバ変数である socket に置かれている。ソースコード 5.4 に受信側となる Local な Queue の Implement に使われるデータ構造を示す。socket は int 型で宣言されている。Implement ファイルで宣言した socket 変数へ socket を保存をすることで、Impl の実装となるプログラム内ならどの CodeGear からでも socket の参照が行える。

Local 側の getData は API としての呼び出しが可能となっている。socket が受け取ったデータを最終的に data として Queue に対して put を行う。Remote 側の sendData は API としては呼び出しは行えず、putAPI が呼び出され Queue への put 処理が終了した後に遷移され、put した Data を接続先の Queue に対して送信する。

ソースコード 5.4: LocalDGMQueue で使われるデータ構造たい

```
typedef struct LocalDGMQueue <> impl CQueue {
  struct Element* top;
  struct Element* last;
  struct Atomic* atomic;
  int* socket;
} LocalDGMQueue;
```

現状ではデータの通信に行われる DataGear は全ての DataGear の Union(共用体) となる Data 型を用いている。送信データのサイズを Union Data 型のサイズにしている理由は、sizeof 関数で返される値は共用体に含まれる構造体の中で最も大きなメモリサイズを

返すためである。これにより put された DataGear の型がどのようなものであれ、受け取り側が DataGear の型を正しく選択している限りはデータの整合性を守ることができる。しかし、受信側が受け取ったデータレコードの正しい型を把握しているとは限らないため、将来的には Java における messagePack に相当する機能になどによるシリアライズしたデータを送信する形にする。Socket の処理中に問題が発生した場合は inputDataGear として指定した__code whenError() として入力された CodeGear に対して遷移する。

ソースコード 5.5: LocaDGMQueue の getData

```
__code getDataLocalDGMQueue(struct LocalDGMQueue* cQueue, __code next
    (...), __code whenEOF(...), __code whenError(...)){
    int recv_size, send_size;
    char send_buf;

    union Data* recv_data;
    recv_size = recv(cQueue->socket, recv_data, sizeof(union Data), 0);
    if (recv_size == -1) {
        printf("recv error\n");
        goto whenError(...);
    }
    if (recv_size == 0) {
        printf("connection ended\n");
        goto whenError(...);
    }


    FileString* fileString = NEW(FileString);
    fileString = recv_data;
    if (fileString->EoF) == 1) {
        send_buf = 0;
        send_size = send(cQueue->socket, &send_buf, 1, 0);
        if (send_size == -1) {
            printf("send error\n");
        }
        close(cQueue->buffer);
        goto whenEOF(...);
    } else {
        send_buf = 1;
        send_size = send(cQueue->socket, &send_buf, 1, 0);
        if (send_size == -1) {
            printf("send error\n");
            goto whenError(...);
        }
    }

    Gearef(context, cQueue)->data = recv_data;
    goto putLocalDGMQueue(recv_data, next);
}
```

ソースコード 5.6: RemoteDGMQueue の sendData

```
__code sendDataRemoteDGMQueue(struct RemoteDGMQueue* cQueue, union Data*
    data, __code next(...), __code whenError(...)){
    char recv_buf;
    int send_size, recv_size;

    send_size = send(cQueue->socket, data, sizeof(union Data), 0);
    if (send_size == -1) {
        printf("send error\n");
        close(cQueue->socket);
        goto whenError();
    }

    recv_size = recv(cQueue->socket, &recv_buf, 1, 0);
    if (recv_size == -1) {
        printf("recv error\n");
        close(cQueue->socket);
        goto whenError();
    } else if (recv_size == 0) {
        printf("connection ended\n");
        close(cQueue->socket);
        goto whenError();
    } else if(recv_buf == 0) {
        printf("Finish connection\n");
        close(cQueue->socket);
        goto whenError();
    }
    goto next(...);
}
```

5.5　GearsOS における DataGearManager

先で述べた Socket 付きの Queue は、DataGear である Queue と Socket が直接結びついているため、一つの DataGear を用いる場合となる最低限の通信しか行えない。複数の DataGear(key) を用いての通信を構成するためには複数の Queue を蓄えることのできるリストを生成し、そのリスト自体が socket を持つ必要がある。

GearsOS では DataGear を保持する Queue のリストとして赤黒木を用いる。赤黒木は平衡二分木であり、木に対する操作時に行われる操作によりノードの階層が平衡化されるそのため各操作の最悪時間計算量 O(logN) となり、二分木の中でも実用性を備えている。

Queue を所持するリストは Tree でなくとも単純な単方向あるいは双方向リストなどでも実現は可能であるが、GearsOS におけるファイルは複数のプロセスからのアクセスが行われ、アクセスが増えるほど DataGear の key の数が増加するため赤黒木での実装を行う。

DataGear は key ごとに作られた Queue に保存される、従って Queue のリストとなる赤黒木には key ごとの Queue がノードとして保持される。図 5.5 にファイルとなる DataGear-

Manager の任意の key に対して put/take 操作を行う際の処理を示す。いずれの場合もリストとなる赤黒木に対して任意の key の Queue の get 操作を行い、その Queue に対して put もしくは take 操作を行う。あくまでこの DGM はファイルに相当するものなので、赤黒木に含まれている Queue は Input/OutputStream と実際にファイルデータを保持しておく mainDataQueue の三つとなる。しかし WordCount 例題の ack や結果出力の通信など用途に用いる DataGear が必要な場合、Stream となる key 以外を用いるために、ファイルに関連する key 以外にも任意に Queue を追加することもできる。任意の Queue を追加したい場合は store となる Tree に対して、Tree が保持していない key へ put が行われた場合は新しくその key を持つ Queue を生成し Tree に持たせる処理を追加すれば良い。

図に示した手順の Data アクセスではファイルの呼び出しなど DataGear の断続的な参照が行われる場合、Tree への探索処理がボトルネックになってしまう可能性が考えられる。これは特定の key の Queue を main プログラムで直接参照できるように、Tree に Queue 自体のポインタを Output させる API を実装することで解決が行える。

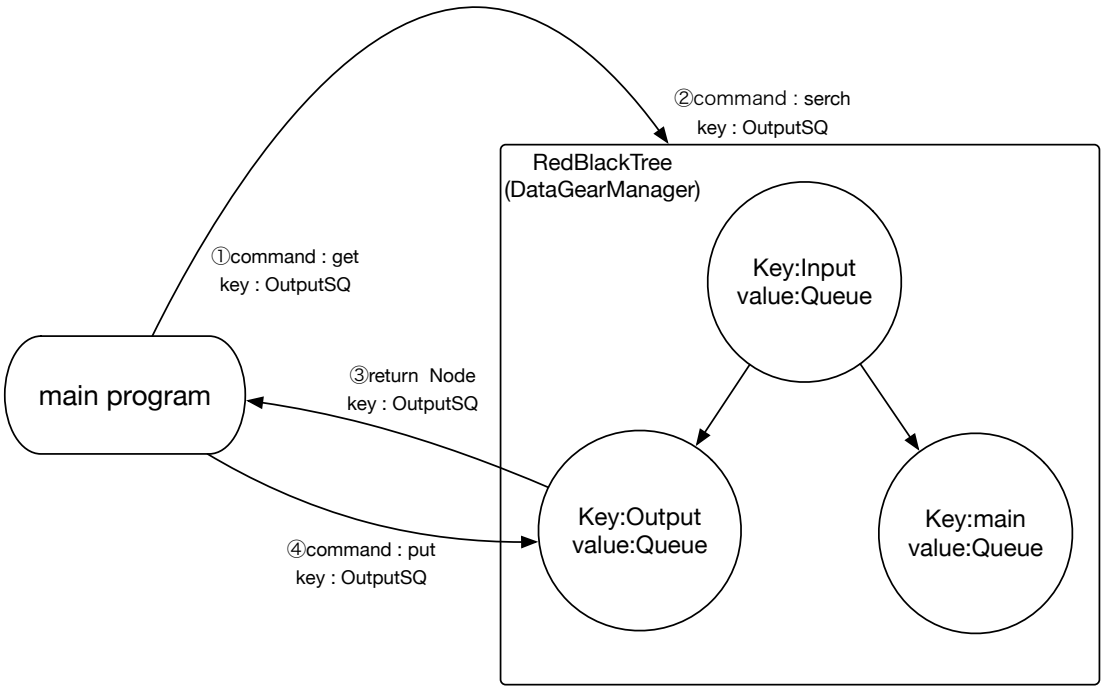

図 5.5: GearsOS におけるファイル (DGM)

5.6　ディレクトリシステム

本研究と並行する形で又吉雄斗による GearsFileSystem のディレクトリ構造の構築が行われている [2]。GearsFS のディレクトリシステムは UnixOS のディレクトリシステムの i-node の仕組みを用いて再現することを試みている。通常の Unix のディレクトリシステムと異なる点として、ディレクトリを赤黒木 (RedBlackTree) を用いて構成する。

図 5.6 に GearsFS の構成図を示す。一つのディレクトリは赤黒木を持ち、あるディレクトリの中に位置するファイル (ディレクトリ含む) は親ディレクトリが持つ赤黒木の中にノードとして同等に配置される。あるディレクトリに存在するファイルを参照する場合は、ディレクトリの持つ Tree に対して探索を行えば良い。図の例では/(ルートディレクトリ) から Users ディレクトリ下にある Dad というディレクトリ (もしくはファイル) を参照するには、/の Tree 内の Users を探索、続いて UsersTree の内部の Dad を探索する形となる。また、親ディレクトリを持つ全てのファイルは、親のディレクトリの情報を保持している。

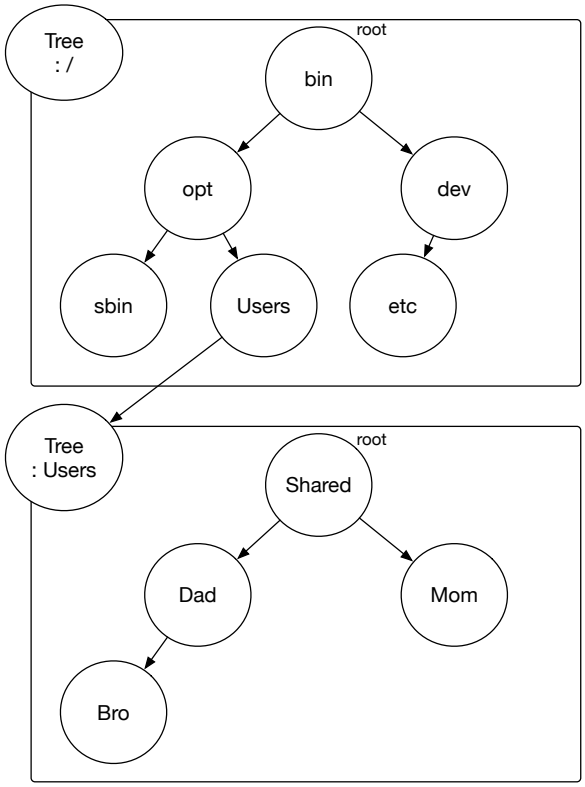

図 5.6: GearsDirectory

Gears のディレクトリシステムは Unix の i-node の仕様を用いる。i-node とはファイル

ごとのユニークな番号 (i-node 番号)。データ領域へのポインタや作成日時、サイズなどのメタデータを保存するための領域である。GearsFS の赤黒木を用いたディレクトリシステムに実際に保存されるノードは key が FileName、value が i-node のペアを保持している。DirectoryTree をファイル名で探索を行うことで任意のファイルの i-node 番号を手に入れ、DirectroyTree とは別に実装された key に i-node 番号とペアとしてファイルのディスクアドレスを保持させる Tree を i-node 番号で探索させる形となる。この形で実装することにより、ファイル自身が所属する Directory を i-node が保持でき、ファイルの複数のアカウントでの共有などによる親 Directory が複数存在する場合や、親 Directory の FileName が変更された場合においても Directory の構造の障害の発生を防げぐことができる。

5.7　GearsFS のバックアップ

GearsOS は将来的に OS 自体がバックアップの機能を保持する構成を目指している。GearsFS の Directory のバックアップは木構造の構築を非破壊的に行うことにより過去のデータを保持する。図 5.7 に非破壊的な木構造の編集を図に示す。非破壊的木構造の編集とは編集元のノードのデータや接続を直接書き換えることなく、根 (ルートノード) から必要なノードは新しく生成し一部のノードは過去のものを使い回すことで更新することを指す。図中の右側の編集後の木構造の赤く記されたノードは新規に作成、もしくはコピーされたノードとなる。変更されたノードと根から変更ノードまでの道筋のノードは、編集元のノードとは別に新しく作成される。それ以外の Tree に属していたノードは編集元の Tree のノードをポインタで接続することで使い回す。つまり、図中の黒いノード 1 から探索を行えば過去のデータのノードが参照でき、赤いノード 1 から探索を行えば編集後の木構造を参照することができる。

ファイル構造体の変更履歴については、ファイルの Data レコードをファイルの変更差分を履歴として保持させる形にすることに加え、変更日時を記録させることで実現したい。diff コマンドようなファイル差分をレコードにすることにより、git や Mercurial のようなバージョン管理を行う。特定の時点のファイルと Directory の状態を呼び出す際は、レコードの変更日時を確認し、参照するべき Tree 構造とレコードの時点まで参照する形となる。

変更ログデータを定期的に任意または一定期間ごとに別のストレージに保存することでバックアップを実現する。また、非破壊的木構造の編集と変更履歴式のデータレコードは変更が行われるたびに、ディレクトリ Tree やファイルの容量増加していくという問題が存在する。この点については任意の期間経過、もしくはユーザの操作により定期的に過去のデータを自動削除もしくは整形することで容量がある程度の削減が望める。また、近年の電子記録媒体の容量増加に伴い問題の重要性が低下していくことが期待できる。

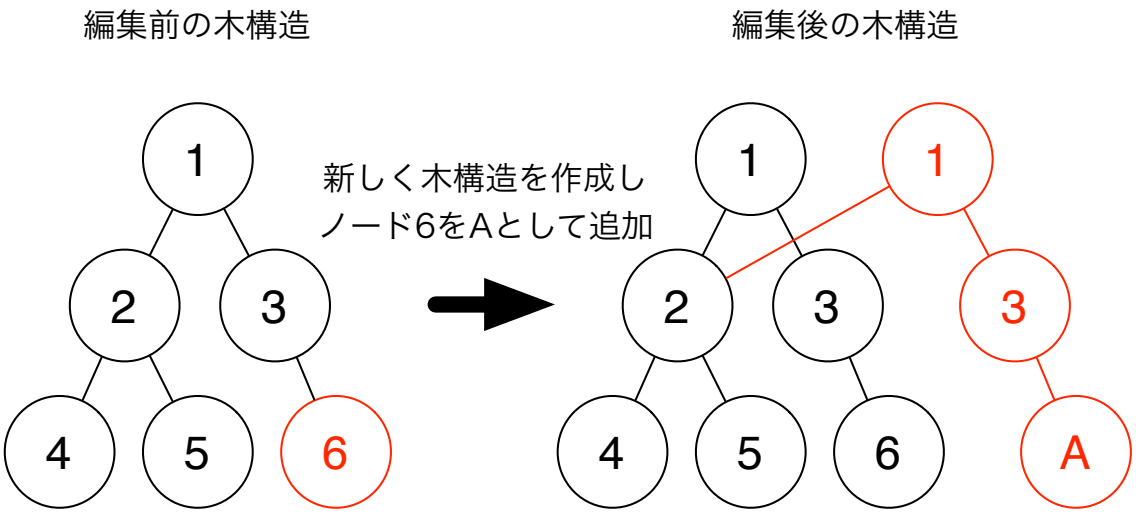

図 5.7: 非破壊的な Tree 編集

5.8　GearsFS の並列処理

分散フレームワーク Christie と GearsOSFileSystem における並列処理について考察した。GearsFS ではファイルは DataGear のリストとして実装される。ファイルの読み取り書き込みを含めた通信は DataGear リストにある特定の key の DataGear の Queue に対して Data を書き込み、もしくは呼び出しを行うことで実現される。そのためファイルは DataGearManager であると言える。

GearsFS のファイルは Christie の DataGearManager の仕組みを参考にするものであるが、Christie の DGM との明確な違いとして、GearsFS の DGM は Christie のように DataGear の差し込みによる通信を構成するだけの用途でなく、ファイルそのものとしての用途を持つという点が存在する。そのため、ChristieDGM のように一つのノード (CGM) により管理されるものではなく、複数のプロセスから競合的にアクセスが行われる DGM である。よって DataGear を保持する Queue は純粋な Queue でなく複数のアクセスが行われても、データの整合性が保たれる Synchronized な Queue を用いる必要が生じる場面がある。

Christie のでは Task(CodeGear) は CodeGearManager により実行が行われる。GearsFS では並列処理を CodeGearManager を用いた実装に代わり、GearsOS に搭載された par goto を利用するという手段が考えられる。par goto とは GearsOS における並列処理構文である。par goto による並列処理では Task を Context で表現し、Task の InputDataGear が揃った Task を Worker で処理を行う。図 5.8 に par goto 構文による並列処理を示す。GearsFS では stream に対する書き込みや取り出しと queue に蓄えられたデータレコードの送受信を並列に実行する。当然複数のファイルが同時に通信される場合はそれらも並行に処理が

行われる。

現状、GearsOS の par goto は実装にいくつか問題点があり、処理速度が比較的遅いことに加え、トランスコンパイラによる書き出しが多くバグが発生しやすい。そのため、par goto による並列処理の構成をより軽量なものに改良する、もしくはファイル通信用の並列処理を独自に開発してしまうという案も考えられる。

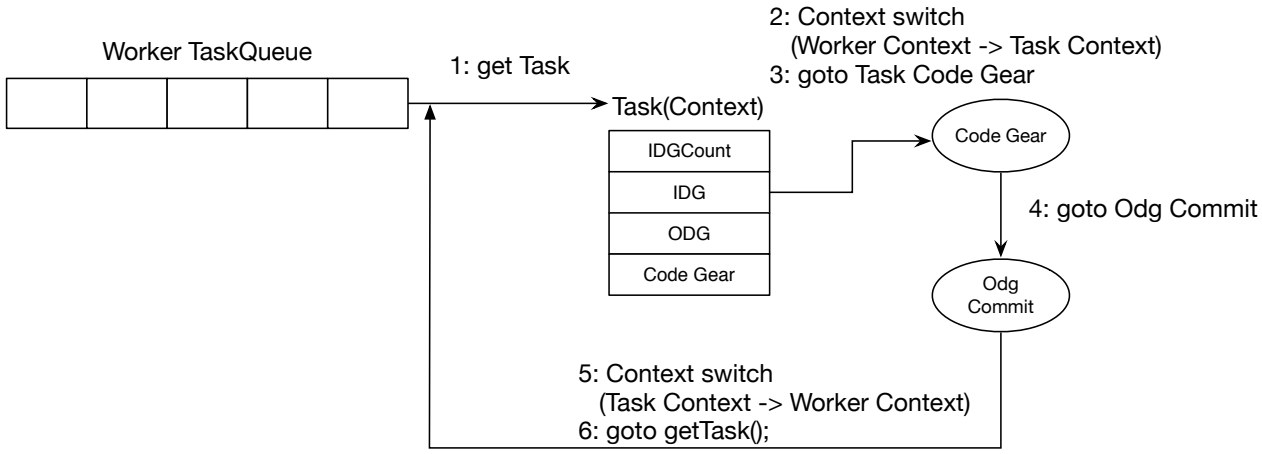

図 5.8: par goto による並列処理

5.9　GearsFS の持続性

GearsFileSystem においてファイルは分割されたデータレコードを保持する Queue を持つリストになることを論じた。メモリ空間上に展開され、変更が行われたデータを SSD などの持続性を持つデバイスへ保存する場合、リスト (DataGearManager) の中の Queue である、ファイルデータそのものを保持している mainDataQueue を格納すれば良い。持続性デバイスは単なるメモリとして扱ってよく、メモリ上のデータ構造と同様に構築する。逆にメモリ上へファイルを展開する場合、デバイス上に保存されている Queue を呼び出し、メモリ上で LocalDataGearManager として構築すれば良い。図 5.9 にデバイス上に保存された DGM を LocalDGM として呼び出す際の遷移図を示す。ストレージ上のブロックのアドレスは unix ファイルシステム同様に、ファイルが保持している i-node が認知していればよい。

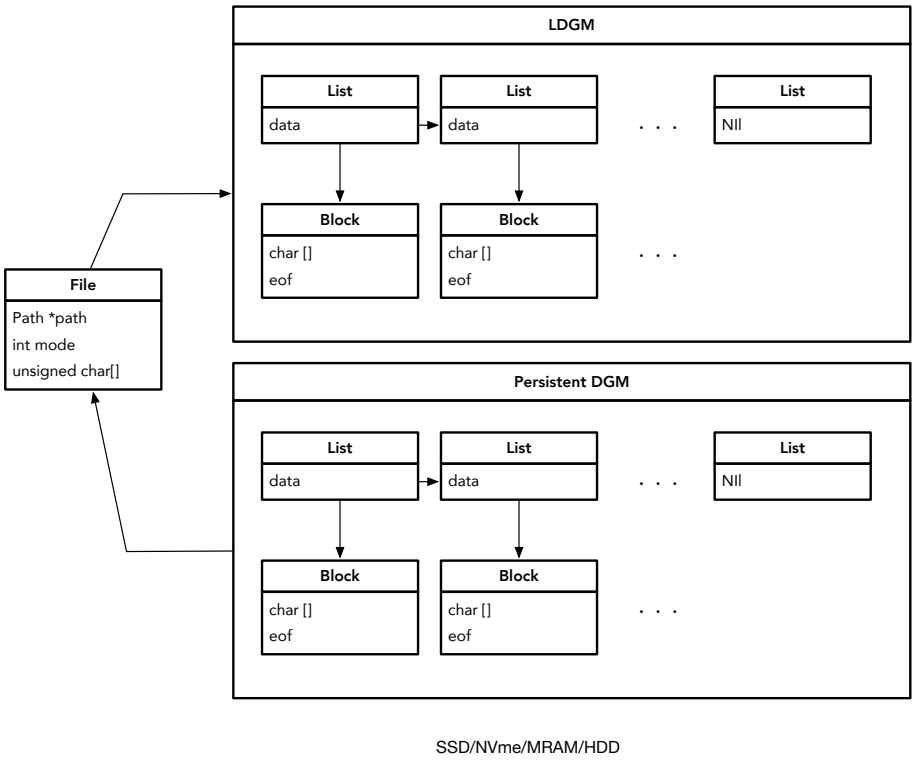

図 5.9: file Persistency

第6章　GearsOSの評価

当研究は純粋なGearsOSを用いて実装されるプロジェクトとしては初めてのものとなっている。本性ではファイルシステムの構築を通したGearsOSの評価を行う。

6.1　トランスコンパイラによるstubCodeGearの誤生成

GearsOSの最大の特徴としてノーマルレベルとメタレベルのプログラムを分離して記述をする必要があるという点が存在する。メタレベルのプログラムは現状、Perlスクリプトを用いたトランスコンパイラにより自動生成されている形となっている。トランスコンパイラにより生成されるメタレベルなプログラムの中で最も重要な部分として、DataGearの受け渡しを担当するstubCodeGearの生成がある。本研究にて行われたGearsOSによるプログラムの記述の上で最も目立った問題点として、stubCodeGearの自動生成部分の仕様とバグが挙げられる。

別のInterfaceを継承したCodeGearからDataGearの受け取りたい場合に、自動生成されたstubCodeGearにバグが生じるという問題点がある。

ソースコード6.1にQueueからDataGearを受け渡しているstubCodeGearとその遷移前と遷移先のCodeGearを示す。このプログラムではTask2にてlocalDGMQueueに対してCodeGear:takeに遷移、Queueのtake処理によって取り出したFileString型のstringをTask3_stubにて呼び出し、Task3へ引き渡している。12行目ではGearefコマンドにてTQueue構造体の中の変数dataをFileString型のstringに格納している。Gearef(context, TQueue)->dataには遷移前のCodeGearにて、take操作でQueueから取り出されたデータが格納されている。このプログラムは正常に動作するが、問題点としてTask3_stubは自動生成が行われたものでなく、手動で記述されていることが挙げられる。

ソースコード6.2にプログラマが記述したものではなく、トランスコンパイラにより自動生成されたエラーのあるTask3のstubCodeGearを示す。この出力されたstubCodeGearの場合、実際にプログラムが実行された場合、Segmentation Faultを起こしてプログラムが終了してしまう。string変数に格納したい、Queueから取り出されたデータの保管場所をstubCodeGearがうまく指定できていないためである。これはFileString構造体はGearsOS内でInterfaceとして見なされており、トランスコンパイラは変数の型を見てGearefコマンドの指定先をContextの保管場所としているために発生しているミスコードとなる。

実際に Queue から取り出されたデータが保存さている場所への参照は Gearef(context, TQueue)->data であるためこれを指定するのが正しい。

ソースコード 6.1: takeCodeGear から遷移される stubCodeGear

```
1  __code Task2(TQueue* localDGMQueue){
2      goto localDGMQueue->take(Task3);
3  }
4
5  __code Task3(TQueue* localDGMQueue, FileString* string){
6      printf("take[%s] [num:%d]\n", string->str, string->size);
7      goto getData();
8  }
9
10 __code Task3_stub(struct Context* context){
11     TQueue* localDGMQueue = (struct TQueue*)Gearef(context, TQueue)->
       tQueue;
12     FileString* string = Gearef(context, TQueue)->data;
13     goto Task3(context, localDGMQueue, string);
14 }
```

ソースコード 6.2: 自動生成にて出力された Task3 の stubCodeGear

```
1  __code Task3_stub(struct Context* context) {
2      TQueue* localDGMQueue = Gearef(context, TQueue);
3      FileString* string = Gearef(context, FileString);
4      goto Task3(context, localDGMQueue, string);
5  }
```

結論としてトランスコンパイラによる自動生成された stubCodeGear は、別 Interface 上の CodeGear から DataGear を継承する場合、どの Interface から DataGear を呼び出せばいいか判断ができないことが分かった。この問題は stubCodeGear をプログラマが記述することにより解決できるが、GearsOS の理想はプログラマが特別な意図を持たない限りは stubCodeGear は記述の必要がないことが望ましい。加えて、CodeGear の処理が同一であっても、遷移前 CodeGear が継承している Interface が異っている場合、複数の CodeGear を記述しなくてはならない。

この問題を解決する案として、Context は stubCodeGear に参照される際に、遷移前の CodeGear が継承している Interface をなんらかの変数などに記憶させるといったことが考えられる。DataGear は基本的に遷移する直前の CodeGear から引き継ぐ場合が多いため、直前の CodeGear の Interface への Gearef コマンドが行える形にすることが望ましい。

6.2　par goto 構文のバグ

GearsFileSystem の API 開発の上で par goto 構文を使った並列処理の利用を試みた。しかし、par goto 構文から自動生成されるメタレベルなコード出力の誤りや Task の終了を

示す__exit が機能しないといった問題が発生した。これらの問題は GearsOS の記述方法に統制が取られていないことが原因として挙げられる。

ソースコード 6.3 に par goto 構文を用いている CodeGear を示す。加えてソースコード 6.4 にソースコード 6.3 をトランスコンパイルした.c プログラムを示す。

この問題点の一つとして Task の終了を示す CodeGear である__exit が機能しないという問題がある。ソースコード 6.3 の 4,5 行目では本体 par goto にて Task:countUp-¿eNum が終了次第、Task を処理した Worker を解放せるための__exit を next_code として指定したい。しかし、下記のコードではトランスコンパイルの時点でエラーを起こし、プログラムのコンパイルが行えない。

6 行目は別の並列プログラム twice での par goto の記述である。異なる点として遷移先の CodeGear がなんらかの Interface の呼び出されたプログラムの中に記述されたものでなく、その CodeGear 内で利用される Interface を InputDataGear として指定していることにある。この記述の場合、__exit は問題なく動作することができるが、GearsOS に実装された Queue や Tree といった多くのプログラムは Interface を継承した上で実装されている。そのため従来のプログラムを Task として利用することが難しくなっている。

加えてトランスコンパイラのバグも確認されている。Interface を継承したプログラム内部へ記述された CodeGear への par goto は InputCodeGear が存在する場合、ソースコード 6.4 の 15, 33 行目の Interface の GET_META が生成されなくなってしまう。_META は並列処理 Task(CodeGear) の inputDataGear の待ち合わせを行う記述である。ソースコード 6.4 の場合、遷移先の Interface の待ち合わせが発生せず、またプリミティブな inputDG となる int 型の 0 を DataGear 名として待ち合わせしようとすることになってしまう。そのため現状、Interface 内 CodeGear の分散処理は行うことができるが、InputDataGear がその Interface 自身と next_code 以外の InputCodeGear が参照できない、加えて__exit が機能しないという問題が存在する。

ソースコード 6.3: takeCodeGear から遷移される stubCodeGear

```
1 __code createTask1(struct TaskManager* taskManager) {
2     struct CountUp* countUp = createCountUpImpl(context);
3     struct CountUp* countUp2 = createCountUpImpl(context);
4     par goto countUp->eNum(0, Task2);
5     par goto countUp2->eNum(0, Task2);
6     //par goto twice(array1, array2, iterate(split), __exit);
7     goto code2();
8 }
```

ソースコード 6.4: 自動生成にて出力された Task3 の stubCodeGear

```
1  __code createTask1(struct Context *context,struct TaskManager*
       taskManager) {
2      struct CountUp* countUp = createCountUpImpl(context);
3      struct CountUp* countUp2 = createCountUpImpl(context);
4
5      struct Element* element;
6                      context->task = NEW(struct Context);
7                      initContext(context->task);
8                      context->task->next = countUp->eNum;
9                      context->task->idgCount = 1;
10                     context->task->idg = context->task->dataNum;
11                     context->task->maxIdg = context->task->idg + 1;
12                     context->task->odg = context->task->maxIdg;
13                     context->task->maxOdg = context->task->odg + 0;
14 ##   GET_META(0)->wait = createSynchronizedQueue(context);
15     //GET_META(countUp)->wait = createSynchronizedQueue(context);
16     Gearef(context->task, CountUp)->countUp = (union Data*) countUp;
17     Gearef(context->task, CountUp)->num = (union Data*) 0;
18     Gearef(context->task, CountUp)->next = C_test;
19                     element = &ALLOCATE(context, Element)->Element;
20                     element->data = (union Data*)context->task;
21                     element->next = context->taskList;
22                     context->taskList = element;
23
24
25     Gearef(context, TaskManager)->taskList = context->taskList;
26     Gearef(context, TaskManager)->next1 = C_code2;
27     goto parGotoMeta(context, C_code2);
28 }
```

第7章　結論

本研究ではGearsOSに導入するファイルの構造、階層表現と分散ファイルシステムのAPIの設計を行った。GearsOSにおけるファイルは競合的なアクセスを許した大域的な資源であり、ファイルは分散フレームワークChristieのDataGearManagerに相当する。ファイル内のデータは任意の構造体によるレコード単位に分割されており、ファイルを読み込む場合、レコードを順番にTakeで呼び出せば良い。ファイルのデータとなるレコードはプログラマが任意の形の構造体で形成することができ、ファイルの種類によって実装を適切なものに構成することができる。

ファイルの共有による通信はRemoteDataGearManagerによるploxyに対して操作することで行われる。そのため、GearsOSでのファイルはファイルそのものであると同時に分散処理の通信とも見ることができる。その通信はWordCount例題の記述によって行い、ChristieAPIの構築とGearsOSにおけるsocket通信の有用性の確認を行った。

また、ファイルは赤黒木によるディレクトリによって階層が構築され、ファイル、ディレクトリは非破壊的な変更が行われる。そのため、変更の履歴はOSが参照できる形で残されており、定期的なバックアップや履歴データの削除などの機能を作成することでアプリケーションに頼らず、OSがバックアップを担当することができる。

加えてファイルシステムの構築をしながら、GearsOS自体の改善点や利用の検証を行った。現状のGearsOSはノーマルレベルとメタレベルの分離という目的を達成してはいるものの、関数スタックを持たない軽量継続という記述の独自な特性や記述の難易度、トランスコンパイラによるバグなどによりプログラマが慣れ親しみ問題を解決しながらプログラムが記述できるようになるまで時間がかかってしまう。GearsOSは現在再実装の検討が行われており、トランスコンパイラによるメタレベルの記述を他の手段を用いるといった議論が行われている。

7.1　今後の課題

ファイルの通信接続はChristieのTopologyManagerの機能を用いるが、詳細な設計は現時点で行えていない。DataGearMangerによる接続形成をTopologyManagerに一任する形にすることで、本来は煩雑な処理が必要となる通信の接続を簡潔に行いたい。しかしChristieのTopologyManagerはあくまで分散フレームワークであるChristieに向けて開発

されたものである。そのため、Christie の TopologyManager を基準にしながら、分散ファイルシステムに向けた仕様を考案していく必要がある。例えば、TopologyManager は参加を表明したノードを任意の形の Topology に構成するというものだが、CodeGearManager の存在しない GearsFS では、RemoteDGM による通信接続に加え、DNS システムによるファイルの呼び出しなど通信 Topology の中枢としての役割を持たせたい。

ファイルシステムの基幹となるディレクトリやファイル通信の設計は行えている。しかし、実際にファイルシステムとして十分な運用が行えるようになるまでの課題は多く存在している。ファイルのアクセス権限への対応やシェルアプリケーションの開発、ディスクスペースの効率の良い配置などはこれからの実装が必要となる。ファイルのメタデータ生成やメモリの効率化などの問題は metaCodeGear 部分で行うことにより、ノーマルレベルのプログラミングでは意識する必要なくプログラミングが行える構成としたい。

また、ファイルシステムの一貫性の保持やキャッシュの設計も行われていない。GearsFS は inode を用いるなど UnixFileSystem を参考とした開発を行っているが、key ストアによるデータ保存形式のため、従来の fsck などの仕組みに代わる機能などの必要性が生じる可能性が高い。

謝辞

本研究の遂行、本論文の執筆にあたり、丁寧な御指導と御教授を賜りました河野真治准教授に心より感謝いたします。ご多忙の中にも関わらず、研究の引き継ぎと御教授を頂きました卒業生である清水隆博さん、赤堀貴一さんに感謝いたします。加えて、共に研究の遂行を行ってくれた又吉雄斗さんを始めとして、共に研究に励み心の支えとなってくださった並列信頼研の所属学生に感謝いたします。最後に、理工学研究科情報工学専攻の教員方、職員方と学友、並びに生活と心の支えをくださった家族に深く感謝いたします。

2022 年 3 月
一木貴裕

参考文献

[1] 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.

[2] 又吉雄斗, 河野真治. Gearsos における i-node を用いた file system の設計. 琉球大学工学部知能情報コース卒業論文, March 2022.

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

[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] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an operating-system kernel. *Commun. ACM*, Vol. 53, No. 6, pp. 107–115, June 2010.

[8] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-button verification of an os

kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pp. 252–269, New York, NY, USA, 2017. ACM.

[9] 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.

[10] Andrew S. Tanenbaum. In *Modern Operating Systems Fourth Edition*, OSDI'15, pp. 265–332, 2015.

[11] 並列信頼研究室. Cbc_gcc. `http://www.cr.ie.u-ryukyu.ac.jp/hg/CbC/CbC_gcc/`. Accessed: 2021-01-31.

[12] 並列信頼研究室. Gearsos. `http://www.cr.ie.u-ryukyu.ac.jp/hg/Gears/Gears/`. Accessed: 2021-01-31.

[13] 並列信頼研究室. Christie. `http://www.cr.ie.u-ryukyu.ac.jp/hg/Database/Christie/`. Accessed: 2021-01-31.

[14] 清水隆博, 河野真治. Gearsos のメタ計算. 琉球大学理工学研究科修士論文, March 2020.

[15] 宮城光輝, 河野真治. 継続を基本とした言語による os のモジュール化. 琉球大学理工学研究科修士論文, March 2018.

[16] 徳森海斗, 河野真治. Llvm clang 上の continuation based c コンパイラ の改良. 琉球大学理工学研究科修士論文, March 2016.

[17] 照屋のぞみ, 河野真治. 分散フレームワーク christie の設計. 琉球大学理工学研究科修士論文, March 2018.

[18] 赤嶺一樹, 河野真治. 分散ネットワークフレームワーク alice の 提案と実装. 琉球大学理工学研究科修士論文, March 2012.

[19] 赤堀貴一, 河野真治. Christie によるブロックチェーンの実装. 琉球大学工学部情報工学科卒業論文, March 2019.

[20] man-page of SOCKET. *https://linuxjm.osdn.jp/html/LDP_man-pages/man2/socket.2.html*.

[21] man-page of GETADDRINFO. *https://linuxjm.osdn.jp/html/LDP_man-pages/man3/getaddrinfo.3.html*.

付 録A　研究会業績

A-1　研究会発表資料

- 分散フレームワーク Christie による Block chain の実装 一木貴裕, 赤堀貴一, 河野真治 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, 2019
- GearsOS の分散ファイルシステムの設計 一木貴裕, 河野真治 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), June, 2021