

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

GearsOS のファイルシステムにおける GC とレプリケー  
ション

GC and Replication in the File System of GearsOS

2024 年 3 月

March 2024

又吉 雄斗

Matayoshi Yuto



琉球大学

大学院理工学研究科

工学専攻知能情報プログラム

Computer Science and Intelligent Systems Course  
Graduate School of Engineering and Science  
University of the Ryukyus

指導教員：教授 和田 知久

Supervisor: Prof. Wada Tomohisa

論文題目:GearsOS のファイルシステムにおける GC とレプリケーション  
氏 名: 又吉 雄斗

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)            長山 格

印  
\_\_\_\_\_  
(副 査)            當間 愛晃

印  
\_\_\_\_\_  
(副 査)            河野 真治

# 要旨

当研究室では, Continuation based C (CbC) を用い, 定理証明やモデル検査などで信頼性を保証することを目的とした GearsOS を開発している. OS の重要な機能の一つにファイルシステムが存在し, GearsOS においても分散ファイルシステムの仕組みや i-node ベースのファイルシステムの実装がされてきた. しかし, 現状の GearsOS のファイルシステムにはデータの多重性を確保するレプリケーションやバックアップの機能や, メモリリークやメモリのフラグメンテーションを解消するガベージコレクション機能が存在しない. よって, これらを GearsOS のファイルシステムのレベルで実装することで, より確実にデータの多重度やメモリの安全性の向上を図ることができると考える. ガベージコレクションやレプリケーションを実装するにはデータをコピーする機能が不可欠である. GearsOS ではデータを RedBlackTree の形式で保持するため, RedBlackTree をコピーすることによってデータのコピーを行うことができる. しかし, GearsOS の RedBlackTree には木をコピーする機能が存在しない. 本研究では, ファイルシステムのレプリケーションやガベージコレクションなどの機能を実装するために必要な RedBlackTree のコピー機能について設計, 構築, 考察とそれを用いたレプリケーションやガベージコレクション機能の設計, 考察を行う. RedBlackTree のコピーでそれぞれの機能を統一的に実装することで, よりシンプルでデータの一貫性が確保された, 定理証明などの形式手法を適用しやすいシステムを実現することを目指す.

# Abstract

In our research lab, we are developing GearsOS, aimed at ensuring reliability through the use of Continuation based C (CbC) for theorem proving and model checking. A critical component of any operating system is its file system, and in GearsOS, there has been implementation of mechanisms for a distributed file system and i-node based file system structures. However, the current GearsOS file system lacks features for data replication and backup to ensure data redundancy, as well as garbage collection functionalities to resolve memory leaks and memory fragmentation. Therefore, by implementing these features at the file system level of GearsOS, we aim to more reliably enhance data redundancy and memory safety. For the implementation of garbage collection and replication, the capability to copy data is essential. In GearsOS, data is stored in the form of a RedBlackTree. Hence, copying the RedBlackTree enables data duplication. However, GearsOS's RedBlackTree currently lacks the functionality to duplicate the tree itself. This research involves the design, construction, and examination of the required RedBlackTree copy functionality for implementing file system replication and garbage collection. Further, we explore the design and examination of replication and garbage collection functionalities using this capability. By uniformly implementing these functionalities through RedBlackTree copying, we aim to realize a simpler system with consistent data, more conducive to formal methods applications such as theorem proving.

# 目次

<b>第 1 章</b>	<b>GearsOS におけるファイルシステムと DB</b>	<b>5</b>
<b>第 2 章</b>	<b>軽量継続を基本とする言語 CbC</b>	<b>7</b>
2.1	Gear の概念 . . . . .	7
2.2	goto による軽量継続 . . . . .	7
2.3	CodeGear の記述例 . . . . .	8
<b>第 3 章</b>	<b>信頼性の保証を目的とした GearsOS</b>	<b>10</b>
3.1	3 種類の GearsOS . . . . .	10
3.2	メタ処理を記述する metaGear . . . . .	10
3.3	全ての Gear を参照する Context . . . . .	11
3.4	モジュール化の仕組み Interface . . . . .	12
3.5	GearsOS の RedBlackTree . . . . .	15
3.6	ALLOCATE . . . . .	18
<b>第 4 章</b>	<b>GearsFileSystem</b>	<b>19</b>
4.1	i-node を用いたディレクトリシステム . . . . .	19
4.2	非破壊 RedBlackTree による構成 . . . . .	21
4.3	RedBlackTree のトランザクション . . . . .	21
4.4	ディスク上とメモリ上のデータ構造 . . . . .	22
4.5	DataGearManager による分散ファイルシステム . . . . .	23
<b>第 5 章</b>	<b>GearsFileSystem における GC とレプリケーション</b>	<b>24</b>
5.1	ファイルシステムの信頼性に関する機能 . . . . .	24
5.2	メモリの管理手法 . . . . .	24
5.3	GearsFileSystem の GC . . . . .	25
5.4	GearsFileSystem のレプリケーション . . . . .	26
5.5	コピー実行のタイミング . . . . .	28

<b>第 6 章</b>	<b>CopyRedBlackTree の実装</b>	<b>30</b>
6.1	Tree Interface の Copy API . . . . .	30
6.2	コピーのアルゴリズム . . . . .	31
6.3	コピーの実装の詳細 . . . . .	32
<b>第 7 章</b>	<b>まとめと今後の課題</b>	<b>34</b>
7.1	ファイルシステムにおけるスキーマ . . . . .	34
7.2	RedBlackTree による権限の表現 . . . . .	35
7.3	データクエリ言語 . . . . .	35
7.4	ログなどの時系列データの保存 . . . . .	35
7.5	スタンドアロンな DB . . . . .	35
7.6	GC タイミングに必要な処理の自動挿入 . . . . .	35
	<b>謝辞</b>	<b>36</b>
	<b>参考文献</b>	<b>37</b>
	<b>研究関連論文業績</b>	<b>39</b>
	<b>付録</b>	<b>39</b>
	<b>付 録 A 研究会業績</b>	<b>40</b>
A-1	研究会発表資料 . . . . .	40

# 目次

2.1	CodeGear と入出力の関係図 . . . . .	8
3.1	CodeGear と MetaCodeGear の関係 . . . . .	11
3.2	Context を参照する流れ . . . . .	12
3.3	RedBlackTree のノードの種類 . . . . .	17
4.1	i-node を用いたディレクトリシステムの処理の流れ . . . . .	20
4.2	トランザクショナルな write 操作 . . . . .	22
4.3	非破壊的な Tree 編集 . . . . .	23
5.1	RedBlackTree の Copy による GC . . . . .	26
5.2	GearsFileSystem のレプリケーションの基本設計 . . . . .	27
5.3	GC 実行処理の挿入 . . . . .	28
6.1	コピー元の Tree と 2 つの Stack の状態の例 . . . . .	32
6.2	Copy 時の CodeGear の大まかな遷移 . . . . .	33

# ソースコード目次

2.1	CbC のプログラム例 . . . . .	8
3.1	Queue のインターフェース . . . . .	12
3.2	Interface の呼び出し . . . . .	13
3.3	Queue のインターフェース . . . . .	13
3.4	SingleLinkedListQueue の型定義 . . . . .	15
3.5	Tree の仕様 . . . . .	15
3.6	RedBlackTree の実装 . . . . .	16
3.7	RedBlackTree の実装の型定義 . . . . .	16
3.8	Node の型定義 . . . . .	17
3.9	ALLOCATE の定義 . . . . .	18
5.1	実行する CodeGear の切り替えのコード . . . . .	29
6.1	Tree Interface の使用定義 (Copy 追加後) . . . . .	30
6.2	RedBlackTree の実装の型定義 (Copy 追加後) . . . . .	30



# 第1章 GearsOSにおけるファイルシステムとDB

情報システムの信頼性を確保することは重要な課題である。2023年には銀行システムや航空機の旅客システム、電子決済システムなどで障害が発生した [1, 2, 3]。信頼性の高いシステムを構築することは、これらのような社会的影響のあるシステムの重大な障害発生防止につながり、サービス提供者や受容者の機会的、経済的損失を防ぐことにつながる。情報システムはアプリケーション、OS、DB、メモリやSSDなどのハードウェア、分散ノードやネットワークなどさまざまな要素で構成される。それらのうちどれか1つでも信頼性を損なうと、システム全体の信頼性の低下につながる。情報システム全体の信頼性を確保するためには、これらの要素それぞれにおいて、信頼性を保証する必要がある。

当研究室では、信頼性の保証を目的としたGearsOSを開発している。GearsOSは、定理証明やモデル検査などの形式手法を用いて信頼性を保証できることを目標としている。[4]。一般的に信頼性を保証する手法としてテストコードを用いることが挙げられる。しかしながら、OSなどの大規模なソフトウェアにおいて人力で記述するテストコードのみではカバレッジが不十分であり、検証漏れが発生する可能性がある。GearsOSではテストコードに加え、形式手法を用いることでより高い信頼性の保証を目指している。GearsOSは当研究室で開発しているプログラム言語であるContinuation based C(CbC)で記述されており、ノーマルレベルとメタレベルを容易に切り分けることを可能とする拡張性を有す。CbCによって本来行いたい処理をノーマルレベルで記述し、信頼性を保証するための処理をメタレベルで記述するといった書き分け、拡張を比較的容易に可能とする。

OSの重要な機能の1つとしてファイルシステムが挙げられる。ファイルシステムはOSのプロセスやユーザーデータの管理に必要な機能である。ファイルシステムには可変長の文字列を格納するファイルと、そのファイルにアクセスするための名前管理の機能がある。ファイルの名前の一貫性は名前管理によって保証される。しかし、ファイルに同時に書き込まれた際の一貫性を保証する機能をファイルシステムとしては持っておらず、ファイルの書き込みを制御するロック機構をアプリケーションが持つことによって、ファイルの一貫性を保証している。ファイルシステムによく似たものとしてDBが挙げられる。DBは入力の属性名と型の組み合わせを複数持つレコードと、特定の属性をキーとしたテーブルがある。また、レコードのinsert, delete, updateの直列化可能性を保証する

機能を持つ。ファイルシステムと DB は格納するものの形式やそれにアクセスする方法、直列化可能性を保証する手法が異なるが、データがある形式で保持する仕組みであるという本質的な部分において違いはない。よって、ファイルシステムと DB を同一のシステムとして実装することが可能であると考えられる。

ファイルシステムは OS において重要な機能であるため GearsOS においても実装をする必要があると考えられ、当研究室では分散ファイルシステムや i-node を用いたファイルシステムの設計がされてきた [5, 6]。しかしながら、データの多重度や一貫性を確保するための機能がないため実装したい。本研究では、GearsOS においてデータの一貫性を確保しつつ、レプリケーションやガベージコレクションの実装するために必要である、RedBlackTree の Copy の仕組みを設計、構築を行った。また、それを用いたレプリケーションやガベージコレクション、バックアップ機能の設計に関する考察を行った。

## 第2章 軽量継続を基本とする言語 CbC

Continuation based C(CbC)[7, 8]はC言語の下位言語であり、関数呼び出しを行わない軽量な継続を基本とするプログラミング言語である。CbCは処理の単位の CodeGear, データの単位の DataGear といった Gear の概念をもつ。CodeGear はC言語などにおける関数と違い、goto による jmp 命令が用いられ、プログラムの継続においてコールスタックを持たない。これを通常の実数による継続と区別して、軽量継続と呼ぶ。軽量継続によってリフレクションのような処理の挿入や切り分けを容易にしている。

### 2.1 Gear の概念

CbC には処理の単位の CodeGear とデータの単位である DataGear という概念が存在する。CodeGear は `_code` という記述で宣言することができる。CbC はC言語の下位言語であるため、通常の実数も使用することは可能だが、基本的に CodeGear の単位でプログラミングを行う。DataGear は CodeGear で入出力される変数データである。図 2.1 は CodeGear と DataGear の入出力の関係を表している。CodeGear は DataGear を複数入力として受け取ることができ、別の DataGear に複数書き込み出力することができる。特に、入力の DataGear を Input DataGear, 出力の DataGear を Output DataGear と呼ぶ。goto で次の CodeGear に遷移する際、Output DataGear を次の CodeGear の Input DataGear として渡すことができる。

### 2.2 goto による軽量継続

CodeGear から次の CodeGear に遷移していく一連の動作を継続と呼ぶ。通常の実数の場合、関数から次の関数へ遷移する時に function call が行われる。function call は前の関数へ戻る場合があり、そのために call stack を保存する。他方、CbC の継続は function call をせずに goto による jmp で行われる。jmp は function call と異なり、call stack による変数などの環境を保存しない。よって、CbC の goto による継続は function call による継続と比較して軽量であるといえる。そのことから、CbC における継続を function call による継続と区別して、軽量継続と呼ぶ。軽量継続の利点としてリフレクションのような処理

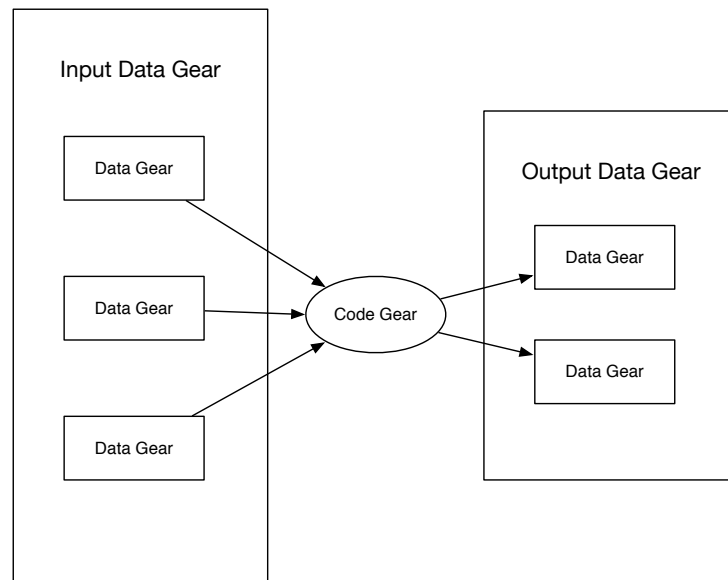


図 2.1: CodeGear と入出力の関係図

をより柔軟に行える点が挙げられる。リフレクションはプログラム自身のメタデータを分析し、それによってプログラムを実行時に書き換える一種のメタプログラミング手法である。一般的にクラスやメソッド、関数の単位で書き換えが行われる。手法の例として Java における AspectJ ライブラリを用いたプログラミングが挙げられる。軽量継続の場合、CodeGear 遷移のどの地点においてもメタな処理を挿入することが可能であるため、より柔軟なリフレクションが可能と考える。

### 2.3 CodeGear の記述例

CbC のプログラム例をソースコード 2.1 に示す。まず main 関数において add1 CodeGear へ goto を行う。その際 add1 へ Input DataGear として  $n$  を渡す。C の goto が *goto label;* という記法で、ラベリングした箇所へ jmp を行うのに対し、CbC の goto は *goto add1(n);* という記法で、add1 CodeGear へ  $n$  DataGear を渡して jmp を行う。add1 は処理の最後に add2 CodeGear へ goto を行う。その際 Output DataGear out\_n を add2 の Input DataGear として渡す。このように CbC では CodeGear の Output DataGear を次の CodeGear の Input DataGear として渡すことを繰り返すことで処理を進める。

ソースコード 2.1: CbC のプログラム例

```

1 | __code add1(int in_n) {
2 |     int out_n = n + 1;
  
```

```
3 |     goto add2(out_n);
4 | }
5 |
6 | __code add2(int in_n) {
7 |     int out_n = n + 2;
8 |     goto end(out_n);
9 | }
10 |
11 | __code end(int in_n) {
12 |     printf("%d", n);
13 | }
14 |
15 | int main(int argc, char *argv[]) {
16 |     int n = 1;
17 |     goto add1(n);
18 | }
```

# 第3章 信頼性の保証を目的とした GearsOS

GearsOS[9, 10, 11] は当研究室で開発している，信頼性と拡張性の両立を目的とした OS である．GearsOS には Gear という概念があり，実行の単位を CodeGear，データの単位を DataGear と呼ぶ．軽量継続を基本とし，stack を持たない代わりに全てを Context 経由で実行する．同様に Gear の概念を持つ Continuation based C (CbC) で記述されており，ノーマルレベルとメタレベルの処理を切り分けることが容易である．また，GearsOS は現在開発途上であり，OS として動作するために今後実装しなければならない機能がいくつか残っている．

## 3.1 3種類 of GearsOS

GearsOS には現在 3 つの種類がある．1 つ目が形式手法による信頼性の向上を目的とした，GearsAgda と呼ばれる GearsOS である [12]．これは，Agda によって実装されており，森 逸汰による GearsAgda による Red Black Tree の検証などの取り組みがされている [13]．2 つ目はスタンドアロン OS の開発を目的とした，CbC\_xv6 と呼ばれる GearsOS がある [14]．これは，教育用に開発された x.v6[15] を CbC で書き換える形で実装している．CbC\_xv6 では仲吉 菜々子による Gears OS の CodeGear Management の取り組みがされている [16]．3 つ目はユーザーレベルタスクマネジメントの実装を目的とした GearsOS がある．これは，CbC によって実装されており，分散ファイルシステムの設計や RedBlackTree でのディレクトリシステムの構築などの取り組みがされている [5, 6]．

本研究では，CbC によって実装されたユーザーレベルタスクマネジメント実装の GearsOS を対象にファイルシステムのレプリケーションや GC 機能の実装を考える．以下，GearsOS はユーザーレベルタスクマネジメント実装の GearsOS を指す．

## 3.2 メタ処理を記述する metaGear

図 3.1 は CodeGear の遷移と MetaCodeGear の関係を表している．OS のプログラムはユーザーが実際に行いたい処理を表現するノーマルレベルと，カーネルが行う処理を表

現するメタレベルが存在する。ノーマルレベルで見ると CodeGear が DataGear を受け取り、処理後に DataGear を次の CodeGear に渡すという動作をしているように見える。しかしながら、実際にはデータの整合性の確認や資源管理などのメタレベルの処理が存在し、それらの計算は MetaCodeGear で行われる。その際、MetaCodeGear に渡される DataGear のことは特に MetaDataGear と呼ばれる。また、CodeGear の前に実行される MetaCodeGear は特に stubCodeGear と呼ばれ、メタレベルを含めると stubCodeGear と CodeGear を交互に実行する形で遷移していく。

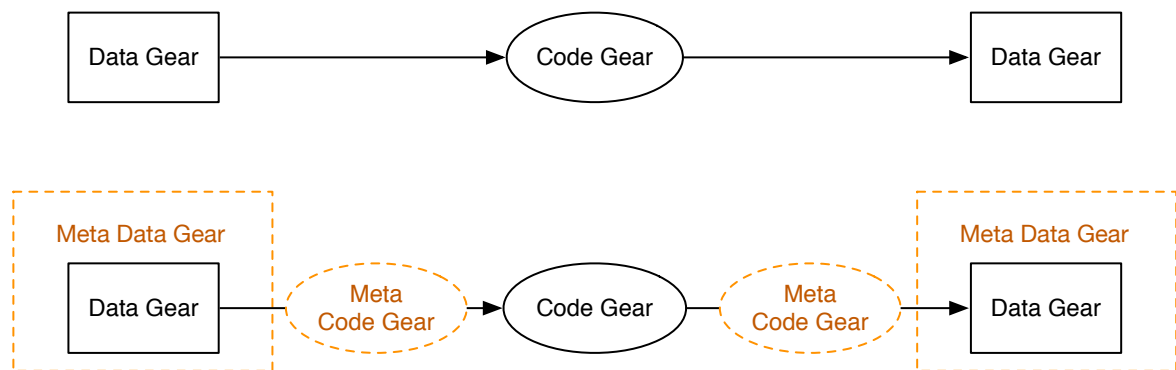


図 3.1: CodeGear と MetaCodeGear の関係

### 3.3 全ての Gear を参照する Context

Context は GearsOS 上全ての CodeGear, DataGear の参照を持ち、CodeGear と DataGear の接続に用いられる。OS 上の処理の実行単位で、従来の OS におけるプロセスに相当する機能であるといえる。また、CodeGear を DataGear の一種であると考えると、Context は Gear の概念では MetaDataGear に当たる。Context はノーマルレベルから直接参照されず、必ず MetaDataGear として MetaCodeGear から参照される。それは、ノーマルレベルの CodeGear が Context を直接参照してしまうと、メタレベルを切り分けた意味がなくなってしまうためである。

図3.2は Context を参照する流れを表したものである。まず CodeGear が OutputDataGear ヘデータを output する。stubCodeGear は InputDataGear (前の CodeGear の OutputDataGear) と OutputDataGear を Context から参照し、次の CodeGear へ goto を行う。CodeGear での処理後、OutputDataGear ヘデータを output する。

Context はいくつかの種類に分けることができる。OS 全体の Context を管理する Kernel Context やユーザープログラムごとに存在する User Context, CPU や GPU ごとに存在す

る CPU Context がある。

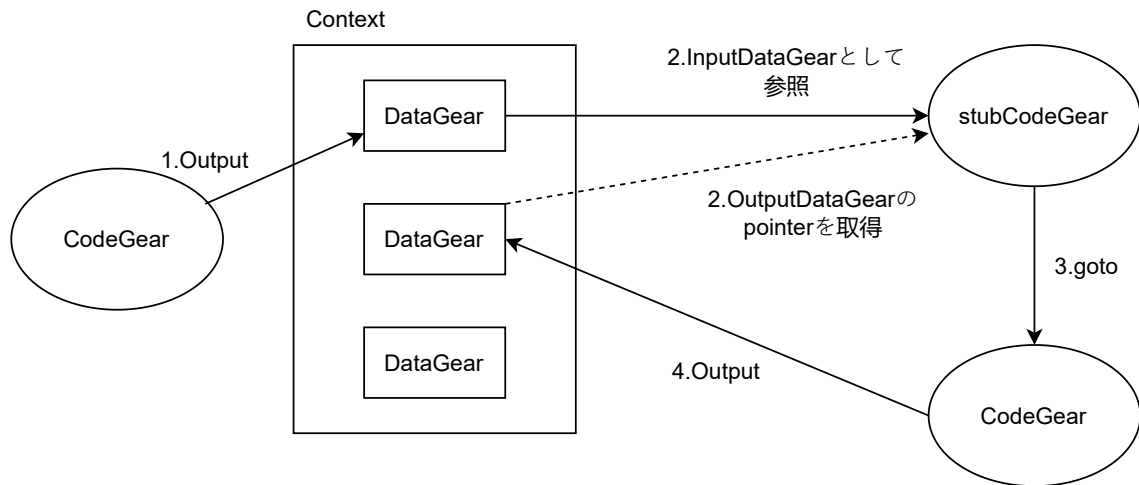


図 3.2: Context を参照する流れ

### 3.4 モジュール化の仕組み Interface

Gears OS にはモジュール化の仕組みである Interface という概念が存在する。モジュール化とは Java のクラスのように複数のメソッドや属性を1つの機能としてまとめて記述することである。GearsOS では Interface によって、DataGear や CodeGear を複数まとめてモジュール化する。Interface は仕様と実装を分けて記述する。例として Queue Interface の仕様記述部分をソースコード 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;
    
```

Interface の仕様は C 言語の構造体定義の形で記述する。2, 3 行目は DataGear を記述しており、DataGear は union Data\* 型で表現する。ここには Interface において、CodeGear



が使用する DataGear を列挙する。5 行目から 10 行目までは CodeGear の引数型を記述しており、`__code` 型で表現する。ここに列挙した CodeGear は Interface の API として機能する。Interface の API の呼び出し例をソースコード 3.2 に示す。

ソースコード 3.2: Interface の呼び出し

```

1  __code odgCommitCPUWorker3(struct CPUWorker* worker, struct Context*
   task) {
2      int i = worker->loopCounter;
3      struct Queue* queue = GET_WAIT_LIST(task->data[task->odg+i]);
4      goto queue->take(odgCommitCPUWorker4);
5  }
```

4 行目で goto によって queue Interface の take CodeGear に継続するよう記述している。take の inputDataGear には odgCommitCPUWorker4 CodeGear を指定している。ソースコード 3.1 の仕様記述では take には queue, next が inputDataGear の型として指定されている。しかし、実際に呼び出す際には next に当たる odgCommitCPUWorker4 のみを渡している。仕様記述の際に全ての CodeGear の第 1 引数 (inputDataGear) に渡している Impl\* queue は、仕様から実装の CodeGear に goto するために必要な記述である。軽量継続において、CodeGear を跨いで状態を保持することはできない。よって仕様から実装に遷移するためには、実装の CodeGear を input DataGear として渡す必要がある。この Impl の第 1 引数は stubCodeGear で自動挿入されるため、実際に API を使用する際は渡す必要がない。inputDataGear の next は CodeGear の処理が終わった際に次に goto する CodeGear を指定する。よって、take CodeGear の処理が全て終了すると、次に odgCommitCPUWorker4 へ goto する。next は next(...) と引数に... が渡される。これは仕様を記述する時点では不定である次に遷移する CodeGear の inputDataGear を表現している。GearsOS で goto する際は実際には Context から必要な値を取り出す。よって、... は必要な値を Context から取り出すことを意味している。

次に Interface の実装について説明する。Queue Interface の実装の 1 つである SingleLinkedListQueue をソースコード 3.3 に示す。

ソースコード 3.3: Queue のインターフェース

```

1  #include "context.h"
2  #include <stdio.h>
3  #impl "Queue.h" as "SingleLinkedListQueue.h"
4  #data "Node.h"
5  #data "Element.h"
6
7  Queue* createSingleLinkedListQueue(struct Context* context) {
8      struct Queue* queue = new Queue();
9      struct SingleLinkedListQueue* singleLinkedListQueue = new SingleLinkedListQueue()
   ;
10     queue->queue = (union Data*)singleLinkedListQueue;
11     queue->take = C_takeSingleLinkedListQueue;
12     queue->put = C_putSingleLinkedListQueue;
```

```

13 |     queue->isEmpty = C_isEmptySingleLinkedListQueue;
14 |     queue->clear = C_clearSingleLinkedListQueue;
15 |     singleLinkedListQueue->top = new Element();
16 |     singleLinkedListQueue->last = singleLinkedListQueue->top;
17 |     return queue;
18 | }
19 |
20 | // 省略~~~~~
21 |
22 | __code putSingleLinkedListQueue(struct SingleLinkedListQueue* queue, union Data*
    data, __code next(...)) {
23 |     Element* element = new Element();
24 |     element->data = data;
25 |     element->next = NULL;
26 |     queue->last->next = element;
27 |     queue->last = element;
28 |     goto next(...);
29 | }
30 |
31 | __code takeSingleLinkedListQueue(struct SingleLinkedListQueue* queue, __code next
    (union Data* data, ...)) {
32 |     printf("take\n");
33 |     struct Element* top = queue->top;
34 |     struct Element* nextElement = top->next;
35 |     if (queue->top == queue->last) {
36 |         data = NULL;
37 |     } else {
38 |         queue->top = nextElement;
39 |         data = nextElement->data;
40 |     }
41 |     goto next(data, ...);
42 | }
43 |
44 | // 省略~~~~~

```

3行目の`#impl as`はInterfaceの実装を記述する際に指定する。implの直後に実装したいInterfaceの仕様を指定し、asの後ろには実装の型名を記述する。であるから、`#impl "Queue.h" as "SingleLinkedListQueue.h"`は仕様Queueの実装をSingleLinkedListQueueとして定義することになる。7行目の`createSingleLinkedListQueue`はSingleLinkedListQueueのコンストラクタを定義しており、DataGearのアロケートやCodeGearを保持するメソッドの初期化を行っている。8, 9行目ではnewでアロケートを行っている。アロケートのようなメタレベルの処理はノーマルレベルには記述されない。そのためこのnewはC言語のnewとは違うGearsOS独自の記述であり、実際にはメタレベルにアロケートを行う処理を挿入している。10~16行目ではSingleLinkedListQueueで使用するCodeGearとDataGearをqueueのメソッドとして初期化している。CodeGearはQueueの仕様で記述したCodeGearと一致している。C\_で始まる記述にはenum CodeにおけるCodeGearの整数を格納している。CodeGearはenum Codeで整数と対応付けられており、この整数を元にCodeGearが参照される。20行目以降では`putSingleLinkedListQueue`や`takeSingleLinkedListQueue`の

ように、仕様で記述された CodeGear を実装している。15, 16 行目では実装の型定義で定義されたその実装独自の DataGear を初期化している。実装の型定義はソースコード 3.4 の通りである。

ソースコード 3.4: SingleLinkedListQueue の型定義

```

1 #include "../ModelChecking/TaskIterator.h"
2
3 typedef struct SingleLinkedListQueue <> impl Queue {
4     struct Element* top;
5     struct Element* last;
6 } SingleLinkedListQueue;

```

3 行目にあるように、実装の型定義では impl キーワードで実装した型名を指定する。4, 5 行目で SingleLinkedListQueue が独自にもつ top, last の DataGear を記述している。

### 3.5 GearsOS の RedBlackTree

Red-black tree(赤黒木) は二分探索木の一種で、ノードに赤か黒の色を付けて色に関するいくつかの条件をもつデータ構造である。木に対する探索, 挿入, 削除操作における最悪計算量が  $O(\log n)$  であるため, 赤黒木は大規模なデータを扱う際に効率的なデータ構造となる。

GearsOS の RedBlackTree は GearsFileSystem で用いられる重要な構造の 1 つであり, ディレクトリ構造を表現するために使用されている。GearsOS における Tree の仕様記述をソースコード 3.5 に示す。

ソースコード 3.5: Tree の仕様

```

1 typedef struct Tree<> {
2     /* future Code */
3     /* Type* tree; */
4     /* Type* node; */
5     union Data* tree;
6     struct Node* node;
7     __code put(Impl* tree, Type* node, __code next(...));
8     __code get(Impl* tree, Type* node, __code next(...));
9     __code remove(Impl* tree, Type* node, __code next(...));
10    // __code clearRedBlackTree();
11    __code next(...);
12 } Tree;

```

ソースコード 3.5 より, Tree は tree DataGear と put, get, remove, next の 4 つの CodeGear を API として持っていることがわかる。他にも探索や木のローテートを行う CodeGear が実装されているが, RedBlackTree の API として提供しているのは put, get, remove の 3 つであり, RedBlackTree Interface の使用者は木に対してこの 3 つの操作ができる。それ

ぞれノードの挿入、取得、削除を行う CodeGear である。取得は、指定した node と一致するノードを木から探し、存在すればそのまま返す。次に、RedBlackTree の実装の記述の一部をソースコード 3.6 に示す。

ソースコード 3.6: RedBlackTree の実装

```

1 #include <stdio.h>
2
3 #include "../context.h"
4 #impl "Tree.h" as "RedBlackTree.h"
5 #interface "Stack.h"
6
7 extern enum Relational compare(struct Node* node1, struct Node* node2);
8
9 Tree* createRedBlackTree(struct Context* context) {
10     struct Tree* tree = new Tree();
11     struct RedBlackTree* redBlackTree = new RedBlackTree();
12
13     tree->tree = (union Data*)redBlackTree;
14     tree->put = C_putRedBlackTree;
15     tree->get = C_getRedBlackTree;
16     tree->remove = C_removeRedBlackTree;
17     // tree->clear = C_clearRedBlackTree;
18
19     redBlackTree->root = NULL;
20     redBlackTree->nodeStack = createSingleLinkedStack(context);
21     return tree;
22 }
23
24 // 省略~~~~~

```

ソースコード 3.6 の 4 行目から、RedBlackTree は Tree の実装であることがわかり、13 ~ 16 行目で仕様に対応する CodeGear を初期化している。19, 20 行目では RedBlackTree が実装で持つ変数を初期化している。次に、RedBlackTree の実装の型定義をソースコード 3.7 に示す。

ソースコード 3.7: RedBlackTree の実装の型定義

```

1 typedef struct RedBlackTree <> impl Tree {
2     struct Node* root;
3     struct Node* current; // reading node of original tree;
4     struct Node* previous; // parent of reading node of original tree;
5     struct Node* newNode; // writing node of new tree;
6     struct Node* parent;
7     struct Node* grandparent;
8     struct Stack* nodeStack;
9     __code findNodeNext(...);
10    int result;
11 } RedBlackTree;

```

2~7 行目は RedBlackTree が持つノードを表しており、それぞれのノードの役割は図 3.3 のように示される。root は RedBlackTree の全てのノードを参照できる親ツリーのルート

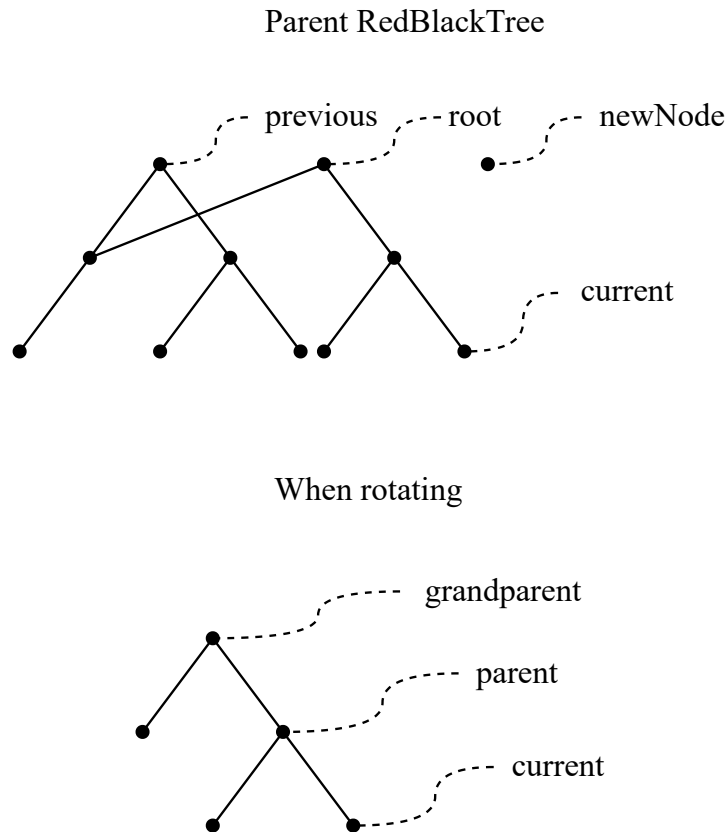


図 3.3: RedBlackTree のノードの種類

ノードを指す。読み込み中のノードは `current` で指されており、追加するノードを `newNode` で表している。 `previous` は木を操作する以前の木を保持するノードであり、これにより非破壊的な操作を可能とする。また、RedBlackTree は挿入、更新、削除の際に木の回転操作を行う。その際、起点のノード (`current`) に対して親のノードを `parent`、祖父母のノードを `grandparent` で指す。8行目の `nodeStack` は木の操作時、木を辿るために使用するスタックである。9行目の `findNodeNext` は `findNode` CodeGear を実行後、次に実行する CodeGear を保持する。10行目の `result` はノードを探索する際のノードの比較結果を保持する。

ソースコード 3.7にある通り、RedBlackTree は Node 構造体を複数持つ。ソースコード 3.8に Node の型定義を示す。

ソースコード 3.8: Node の型定義

```

1 typedef struct Node <> {
2     int key; // comparable data segment;
3     union Data* value;
4     struct Node* left;

```

```
5 | struct Node* right;
6 |     // need to balancing
7 | enum Color {
8 |     Red,
9 |     Black,
10 |     // Red eq 0,Black eq 1. enum name convert intager.
11 | }color;
12 | } Node;
```

1, 2 行目よりノードのキーが int 型であり, value として DataGear のポインタを持つことがわかる. また, 3~7 行目より left, right で子の Node のポインタを持つことによって木構造を構築し, enum Color で RedBlackTree として必要なノードの色を表現していることがわかる.

## 3.6 ALLOCATE

ソースコード 3.9: ALLOCATE の定義

```
1 | #define ALLOCATE(context, t) ({ \
2 |     context->heap = __builtin_align_up(context->heap + sizeof(Meta) ,
3 |     sizeof(void *)) - sizeof(Meta); \
4 |     Meta* meta = (Meta*)context->heap;\
5 |     context->heap += sizeof(Meta);\
6 |     union Data* data = context->heap; \
7 |     context->heap += sizeof(t); \
8 |     meta->type = D_##t; \
9 |     meta->size = sizeof(t); \
10 |     meta->len = 1;\
11 |     meta->data = data; \
12 |     *context->metaData = meta; \
13 |     context->metaData++; \
14 |     data; })
```

## 第4章 GearsFileSystem

ファイルシステムはOSにおいてユーザーやアプリケーションが使用するファイルやプロセスの管理に用いられる重要なシステムである。そのため、GearsOSにおいても i-node を用いたディレクトリシステムや、DataGearManager による分散ファイルシステムの仕組みをもつ、GearsFileSystem の取り組みがいくつかされてきた。

### 4.1 i-node を用いたディレクトリシステム

GearsFileSystem には i-node を用いたディレクトリの仕組みが存在する [6]。i-node は主に Unix 系のファイルシステムで用いられる、ファイルの属性情報が書かれたデータである。inode におけるファイルの属性情報は表 4.1 のようなものがある。また inode は識別番号として inode number を持つ。inode number は一つのファイルシステム内で一意の番号であり、`ls -li` コマンドで確認可能である。inode はファイルシステム始動時に inode 領域をディスク上に確保する。そのため inode number には上限があり、それに伴いファイルシステム上で扱えるファイル数の上限も決まる。inode number の最大値は `df -li` コマンドで確認可能である。

File Types	directory や regular file など、ファイルの種類
Permissions	read write execute の実行可否
UID	ファイル所有者の ID
GID	ファイル所有グループの ID
File Size	ファイルのサイズ
Time Stamps	ファイル作成、編集日時
Number of link	ハードリンクの数
Location on hard disk	データのアドレス

表 4.1: inode でのファイル属性情報

GearsFileSystem では i-node を i-node number が key、i-node でのファイル属性情報を value であるノードを持つ inode tree を RedBlackTree で表現している。また、ファイル名から i-node number を検索するための index tree も同じ RedBlackTree で表現してい

る。データ構造に RedBlackTree のみを用いているのは、ls、cd、mkdir といった、ディレクトリ操作を行うための Unix Like なユーザーインターフェースをもつ。図 4.1 に Gears Directory の処理の例を示す。

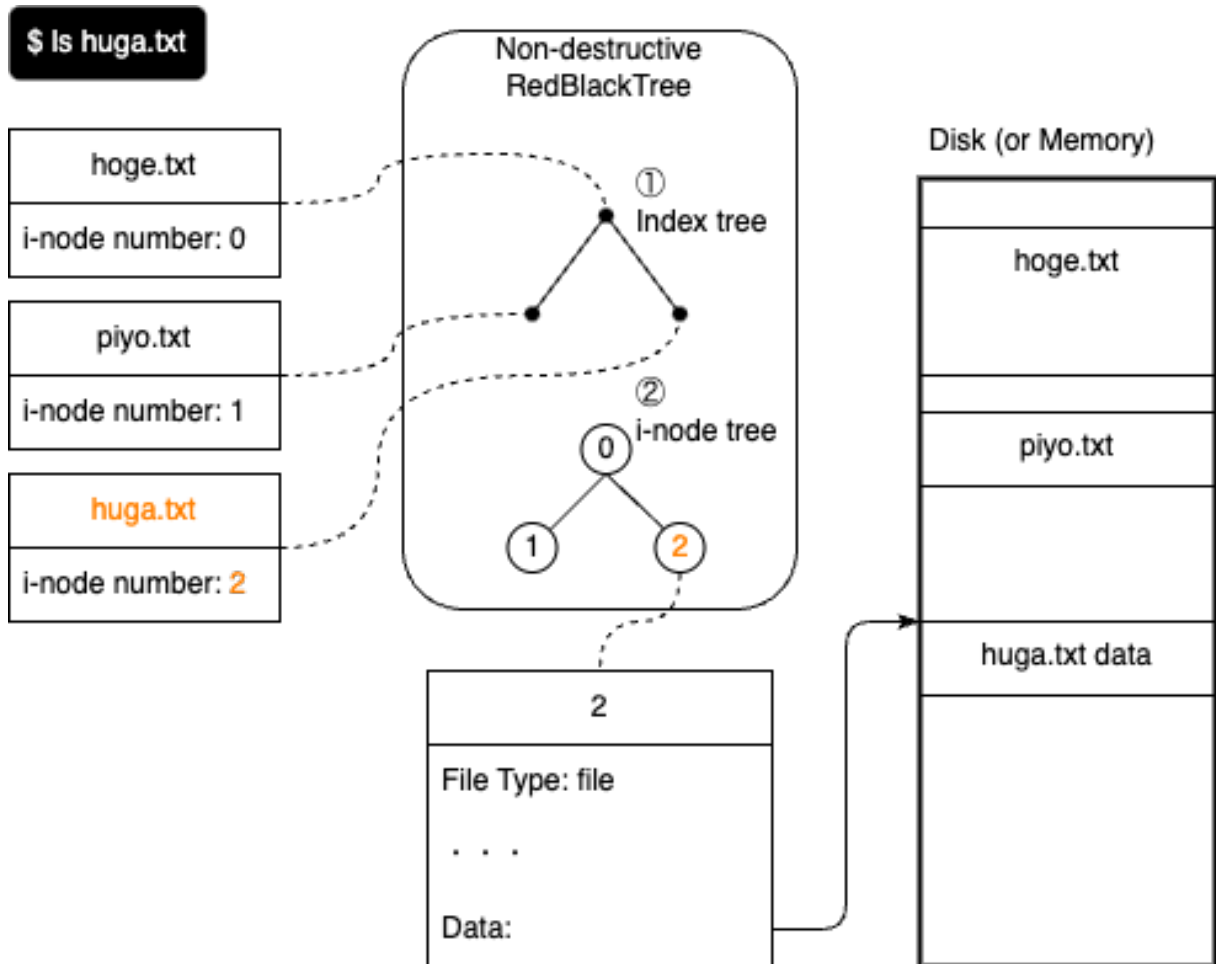


図 4.1: i-node を用いたディレクトリシステムの処理の流れ

ls コマンドはディレクトリ内のファイルやファイル自体の情報を出力するコマンドである。ls hoge.txt を実行すると①index tree を参照し、ファイル名 hoge.txt から i-node number の 2 を取得する。次に、②i-node tree を参照し、i-node number を元にファイルの属性を取得し、ls コマンドの場合はそれを出力する。



## 4.2 非破壊 RedBlackTree による構成

ディスク上とメモリ上でデータの構造は、RedBlackTree に統一する。一般的に、ディスク上のデータ構造として B-Tree が用いられることが多い。なぜならば、HDD を用いる場合はブロックへのアクセス回数を減らしデータアクセスの時間を短くするために、B-Tree のようなノードを複数持つことができる構造が効果的だからである。その点では RedBlackTree は B-Tree に劣る。しかしながら、SSD はランダムアクセスによってデータにアクセスするため、RedBlackTree でなく B-Tree を用いる利点は少ないと考える。よって、ディスク上とメモリ上のデータ構造を RedBlackTree に統一することが考えられる。そうすることによって、ディスク上とメモリ上のデータのやりとりは単純なコピーで実装できる。

## 4.3 RedBlackTree のトランザクション

トランザクションは DB の重要な機能の一つである。データの競合を防ぎ信頼性を向上させ、DB としても扱えるようトランザクションの仕組みを考える必要がある。今回、ファイルシステムは全て RedBlackTree で実装するため、RedBlackTree のノードに対するトランザクションを考える。

トランザクションを write と read に分けて考える。write する場合、トランザクションは RedBlackTree のルートの置き換えと対応する。write するために、まずルートを生やし書き込みが終わった後ルートを置き換える。そのため、書き込みの並列度はルートの数と一致する。しかしながら、ルートの置き換えは競合的なので、複数プロセスから同時に書き込みを行っても 1 つしか成功しない。よって、単一の RedBlackTree に複数の書き込みポイントを作り、並行実行可能にする必要がある。

RedBlackTree に複数の書き込みポイントを作るために、キーごとのルートを作成する。ノードはそれぞれがキーと RedBlackTree を持つ状態になる。write する際は、そのキーのルートを置き換える。それによって、RedBlackTree は複数の書き込みポイントを持つことができ、write を並行実行することが可能となる。

図 4.2 にトランザクショナルな write 操作を表す。A の木はファイルシステム全体を表す RedBlackTree である。ノード N のデータに対して書き込みすることを考えると、キーが a である B の木のルートからロックし C の木を作成して、B の木から C の木のルートに入れ替えることで書き込みを行う。この書き込みを行っている際、A の木のノードはロックしないので A の木のどのノードに対しても並行して書き込み可能となる。

read はデータに変更を加えないため、複数同時に同じノードを読み込むことが可能である。しかし、常に最新の情報を読み込めるとは限らない。最新の情報が欲しい場合は書き込みを一旦止めるような処理が必要になる。

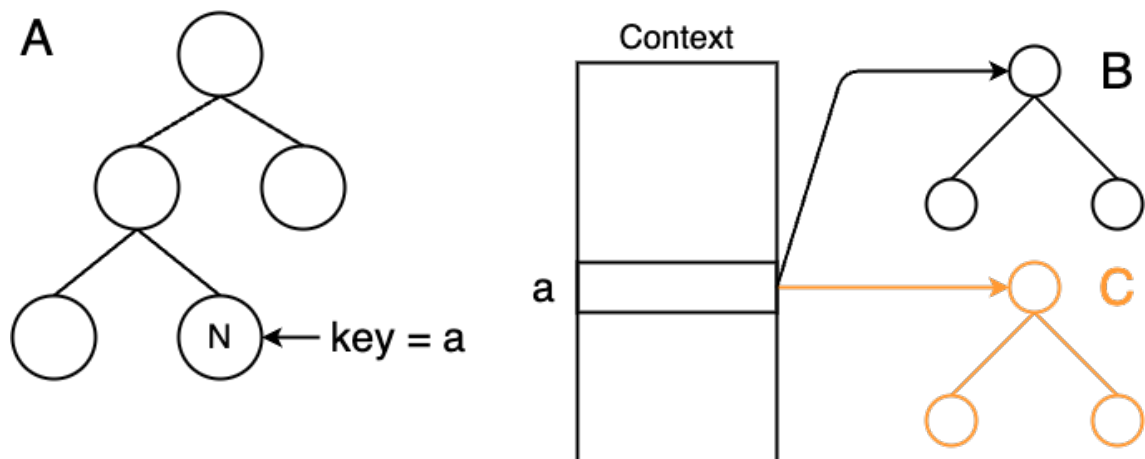


図 4.2: トランザクショナルな write 操作

#### 4.4 ディスク上とメモリ上のデータ構造

ファイルシステムは全て RedBlackTree で構成する。それにより、プログラムの証明がより簡単になるからである。また、ファイルシステムと DB はデータを保管するという本質的な役割は同じである。よって、それらはまとめて RedBlackTree で構成する。

ファイルシステムと DB の違いとして、スキーマが挙げられる。DB は事前にスキーマを定義し、それに沿ってデータを挿入、参照する。対して、ファイルシステムにはスキーマに当たるものがなく、データはファイルパスによって管理される。スキーマを定義することによってデータは構造化され、構造化されたデータは非構造化データと比較して、インデックスを作成することが容易であり、データの検索性が向上する利点がある。しかしながら、スキーマは DB の運用中に変更されることがあり、スキーマを変更した以前へロールバックができない問題がある。

ロールバックがスキーマの変更によって出来なくなることは信頼性に問題があると考えると、スキーマを定義する必要のないスキーマレスな DB が必要になる。ファイルシステムはスキーマレスな DB といえるので、ファイルシステムを構築しつつ DB がスキーマによって実現していた機能を付け加えることを試みる。

RedBlackTree は実装によって、操作が破壊的なものとそうでないものがある。今回用いるのは、非破壊的な実装の RedBlackTree である。図 4.3 は非破壊的編集を行う RedBlackTree を表している。編集前の木構造の 6 のノードを A にアップデートすることを考える。まず、ルートノードからアップデートしたいノード 6 までをコピーする。その後、コピーしたもののノード 6 を A にアップデートする。これにより、アップデート前のノード 6 を破壊することなく A にアップデートが可能である。参照する時は、黒のルートノードから辿れば古い 6 が、赤のルートノードから辿れば新しい A が見える。この仕組みは、デー

タのバックアップやDBのロールバックに用いることが可能だと考える。

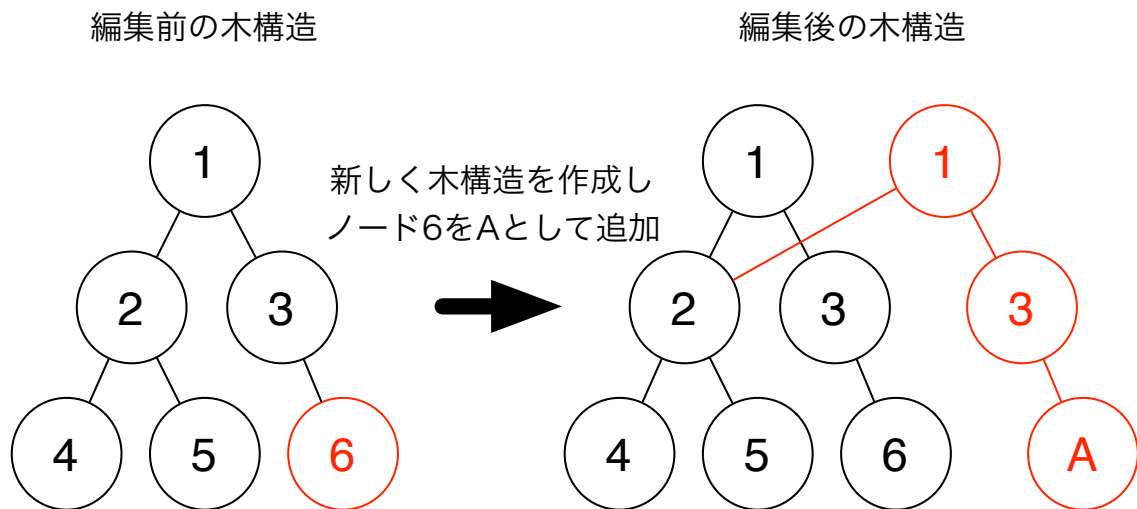


図 4.3: 非破壊的な Tree 編集

## 4.5 DataGearManager による分散ファイルシステム

# 第5章 GearsFileSystemにおけるGCとレプリケーション

本章ではRedBlackTreeのCopyによるGCとレプリケーションの基本設計を述べる。GC、レプリケーション、バックアップをRedBlackTreeのコピーを基礎とする、統一的な仕組みにより実装することが考えられる。

## 5.1 ファイルシステムの信頼性に関する機能

ファイルシステムはデータを保持することを基本的な機能としている。信頼性に関する機能など、その他の機能は追加機能として実装する。ファイルシステムやDBにおける信頼性に関する追加機能として、システムの電源断時にデータが残る persistency, データを書き込めたかどうかを判定する atomic write, 1つのノードが失われた際にデータを保護する多重性, 複数のコピーを調停するコミット機構などが挙げられる。

現状のGearsOSには分散ファイルシステムの通信機能やUnix Likeなインターフェースを持つi-nodeファイルシステムの基本機能は存在するものの、多重性やメモリ管理などの信頼性を確保するための機能が存在しない。データの多重度を確保するための一般的な手法として、データのバックアップやシステム自体のレプリケーションをすることが挙げられる。メモリ管理の機能としてはガーベージコレクションが挙げられる。ガーベージコレクションは通常プログラム言語のレイヤで行われる。これらの機能をファイルシステムのレベルで実装することで、より信頼性の高いファイルシステムを構築したい。

## 5.2 メモリの管理手法

GCのアルゴリズムは大きく分けてMark & Sweep GC, Reference counting GC, Copying GCの3つの種類が存在する。Mark & Sweep GCはマークフェーズとスイープフェーズからなる。マークフェーズはヒープ上でルートから参照することができるオブジェクト全てにマークをし、その後、スイープフェーズでマークされていないオブジェクトを使用されていないオブジェクトのリストであるフリーリストに接続することでGCを行う。

Reference counting GCはオブジェクトの被参照数を表すReference counterを用いるGCである。新たに参照される度にReference counterをインクリメントし、参照が外れる度にデクリメントする。そのようにして、カウンタが0になった時点でフリーリストに接続することでGCを行う。CopyingGCはメモリ上のヒープ領域をFrom領域とTo領域に分割し、ルートから参照できるオブジェクトをFrom領域からTo領域にコピーする。From領域を参照していたポインタはTo領域のオブジェクトを参照するように置き換える。その後、From領域とTo領域を入れ替えることでGCを行う。

一般的にこれらのGC手法は複数を組み合わせて用いられる。世代別GCではオブジェクトの生存期間によって適用するGCアルゴリズムを使い分ける。アロケートされてすぐのオブジェクトを新世代オブジェクト、任意の回数のGCを生き残ったオブジェクトを旧世代オブジェクトとし、それぞれの特性に合ったGCアルゴリズムを適用する。すぐに回収されることが多い新世代オブジェクトはCopying GCで網羅的にGCをし、長く生き残る旧世代オブジェクトはMark & Sweep GCで適宜回収するなどが例として挙げられる。このように複数のGCアルゴリズムを組み合わせることで、それぞれのアルゴリズムの利点を享受できる。

また、メモリ管理手法としてRust言語の所有権がある。所有権ではメモリを所有する変数がスコープを抜ける時に、同時にメモリも解放する。そのためRustではGCの仕組みを必要とせず、より高速にメモリの管理を行うことができる。

### 5.3 GearsFileSystemのGC

GearsFileSystemのGCはCopying GCを基本的なアルゴリズムとする。他のGC手法と比較して参照できるオブジェクトをコピーするだけであるため、実装が簡単で、より高いスループットが期待できる。Mark & Sweep GCやReference counting GCの場合は、GCを複数フェーズで実装したり、カウンタの扱いについて考える必要がある。また、同様の構造をコピーするのみで実装することによって、データの透過性の確保がしやすい。ファイルやディレクトリを表現するRedBlackTreeは全てのデータの参照を持つ。そのため、オブジェクトルートからオブジェクトを辿ってコピーを行うCopying GCとの相性が良い。

一般的なCopying GCではFrom領域上のオブジェクトをTo領域にコピーする形で実装される。一方、GearsFileSystemではファイルやディレクトリの基本構造であるRedBlackTreeをコピーする。ファイルやディレクトリの操作を行うFromのRedBlackTreeから、ルートから辿れるノードのみをToのRedBlackTreeとしてコピーする。それにより、辿れなかったノード、つまり参照されていないノードはコピーされず、不要なオブジェクトが回収された状態となる。

DBの重要な機能の一つにロールバックがある。RDBのロールバックは、コミットす

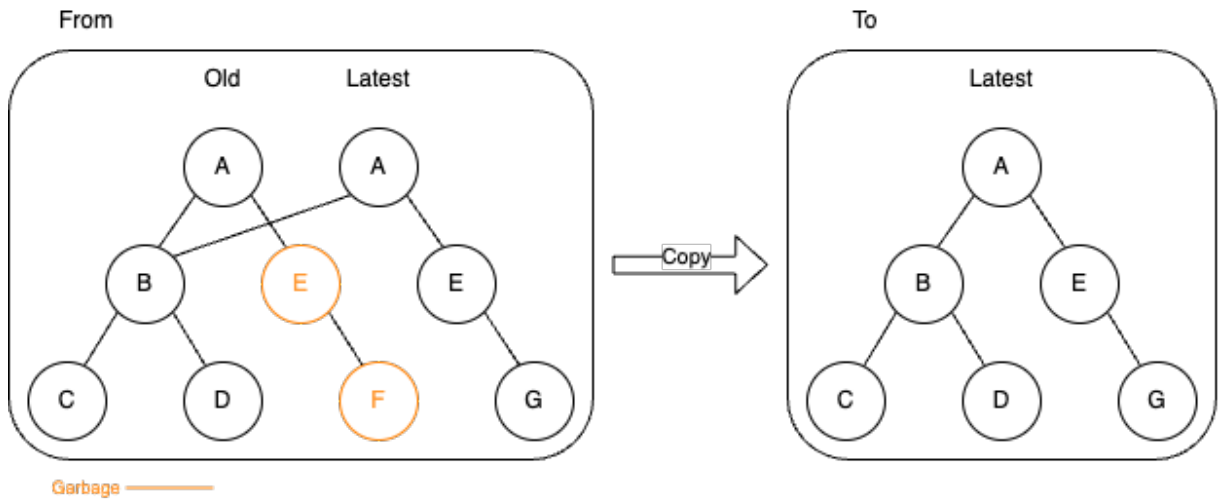


図 5.1: RedBlackTree の Copy による GC

るまではトランザクションの開始時点に戻ることができる機能を持つ。コミットが完了するとそれ以前の状態に戻すことはできないが、データのバックアップをとっておくことで復元を行う。

RedBlackTree のルートノードがデータのバージョンの役割を果たしていることを利用し、データの復元が行える仕組みを考える。非破壊的な Tree 編集はアップデートのたびに、ルートノードを増やす。つまり、ルートノードはアップデートのログと言えその時点のデータのバージョンを表していると考えられることができる。よって、ロールバックを行いたい場合は参照を過去のルートノードに切り替える。

ルートノードはデータのアップデート時に増えるため、データが際限なく増加していく問題がある。この問題は CopyingGC を行うことによって解決する。まず、RedBlackTree を丸ごとコピーして最新のルートを残して他のルートは削除する。その後、コピーしたものはバックアップないしログとしてディスクに書き込む。そうすることで、データの増加によるリソースの枯渇を防ぎ、かつデータのログ付きバックアップを作成することで信頼性の向上が期待できる。

## 5.4 GearsFileSystem のレプリケーション

DB の多重性を確保する機能としてレプリケーションがある。レプリケーションはデータや状態が等しいレプリカを別のノードに生成する機能である。レプリケーション機能があることによって、地理的に距離のあるノードにレプリカを設置することが可能になり、

災害などによって発生するシステム障害を防止することや、アクセス分散によるネットワーク負荷の低減につながる。そのため、GearsFileSystem においてもレプリケーションの機能を実装したい。

GearsFileSystem ではディレクトリを RedBlackTree によって構成しており、RedBlackTree から全てのデータにアクセス可能である。よって、RedBlackTree のコピーを行うことによって、FileSystem のレプリカを作成することが可能であると考えられる。GearsFileSystem のレプリケーションの基本設計を図 5.2 に示す。

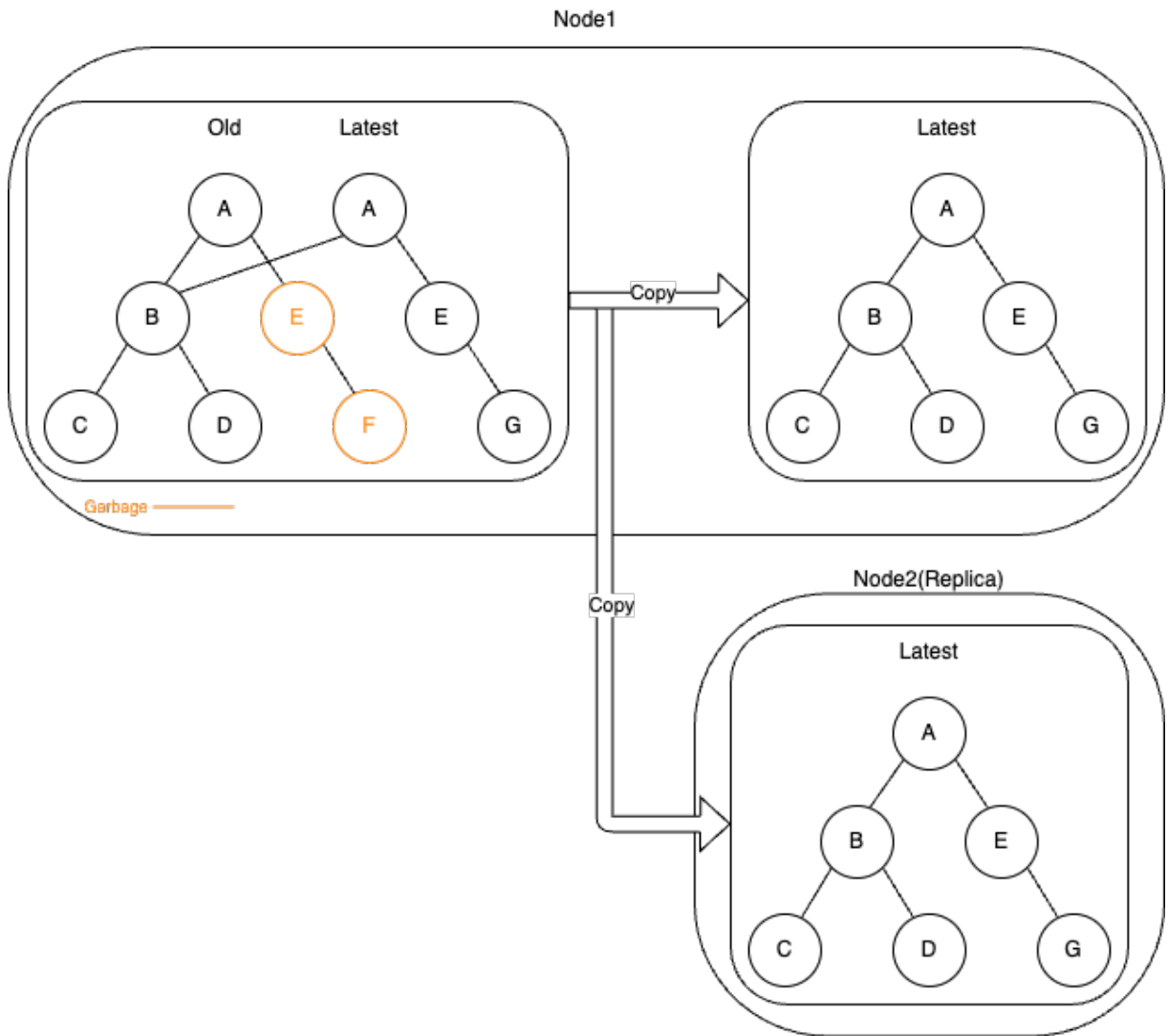


図 5.2: GearsFileSystem のレプリケーションの基本設計

基本的には図 5.1 に示されている GC の仕組みと同様で、RedBlackTree の Copy で構

成される。レプリケーションにおいてコピー元をメイン、コピーをレプリカと呼ぶ場合、Node1がメインでNode2がレプリカとなる。Node1はGCの仕組みと同様であり、違いはNode2にもコピーを行う部分である。このように、GCとレプリケーションを同様の仕組みで実装することが考えられる。しかし、Node2にコピーを行う際はNode1とNode2間で操作やデータを送信するための通信の仕組みが必要である。レプリケーションを実装する際は通信の仕組みについて考える必要がある。

既存のDBにおけるレプリケーション手法は同期のタイミングやレプリカの作成単位によっていくつか種類がある。

### 5.5 コピー実行のタイミング

TODO: ここは Context レベルに書き直す

GCやレプリケーション、バックアップはそれを実行するタイミングが重要である。GCはメモリの使用状況に応じて実行される。例えば、RedBlackTreeに新規ノードを追加する際にメモリが不足した場合などがGCを実行するタイミングとして挙げられる。レプリケーションは同期のタイミングがあり、同期型、非同期型、準同期型が挙げられる。また、バックアップは1日に1回、週に1回など定期的な実行をする。そのため、RedBlackTreeのコピーにコピー実行のタイミングを制御する機構が必要である。GearsOSは拡張性の高いCbCによって記述され、本来実行したい処理に追加の処理を挿入することが比較的容易に可能である。図5.3にGC実行処理の挿入の仕組みの例を示す。

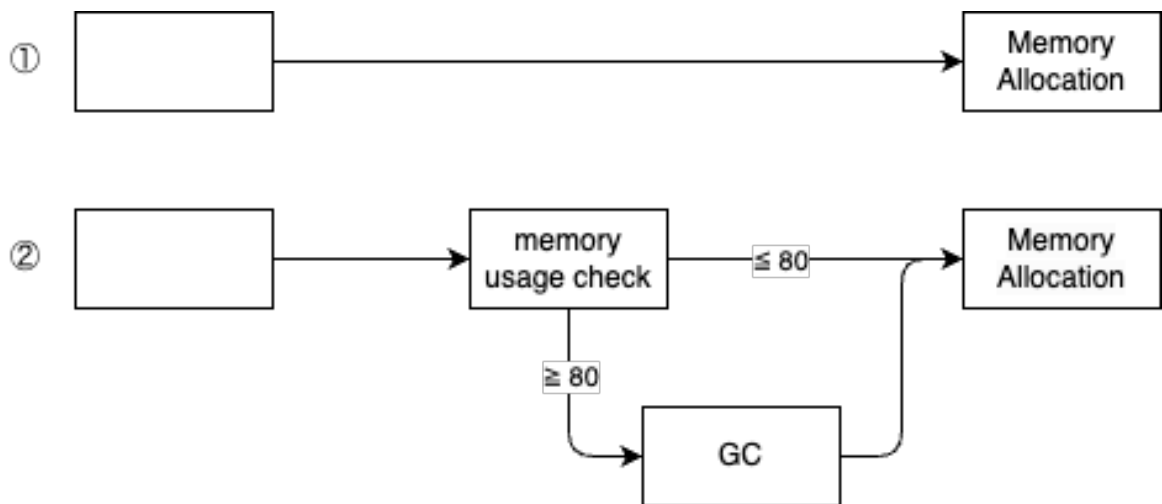


図 5.3: GC 実行処理の挿入

矢印は軽量継続であり、①ではメモリアロケーションを必要とする本来行いたい処理



を表している。②はメモリ使用率をチェックする CodeGear を挿入し、使用率が 80% 以上の場合に GC を実行する。メモリ使用率をチェックする CodeGear は条件によって実行する CodeGear を切り替えるものであり、既存の GearsOS のコードにもこのような処理を行うものは存在し、例として SingleLinkedStack のコードの一部をソースコード 5.1 に示す。

ソースコード 5.1: 実行する CodeGear の切り替えのコード

```
1 __code isEmptySingleLinkedStack(struct SingleLinkedStack* stack, __code
  next(...), __code whenEmpty(...)) {
2     if (stack->top) {
3         goto next(...);
4     } else {
5         goto whenEmpty(...);
6     }
7 }
```

CodeGear は遷移先の CodeGear の参照を持つ必要があるため、inputDataGear として next と whenEmpty を渡している。条件分岐は通常の if 文で行われ、条件ごとに次の CodeGear を指定している。

このようにして、条件分岐をして実行する CodeGear を切り替えることができる。しかし、GC は本来行いたい処理ではなくメタレベルの処理である。そのため、GC への切り替えにおいてソースコード 5.1 のようなコードを記述すると、ノーマルレベルとメタレベルが混在する CodeGear となってしまう問題がある。

## 第6章 CopyRedBlackTreeの実装

データのバックアップやレプリケーション, GCの機能を実装するためには, データのコピーをする必要がある. GearsOSのファイルシステムにおいて, データは全てRedBlackTreeに格納される. しかしながら, 現状のRedBlackTreeには木をコピーする機能が無い. よって, RedBlackTreeに木のコピー機能を実装する必要がある. CopyRedBlackTreeの実装について述べる.

### 6.1 Tree InterfaceのCopy API

CopyRedBlackはTree InterfaceのAPIの1つとして実装した. ソースコード6.1にCopy APIを追加したTree Interfaceの仕様定義を示す.

ソースコード 6.1: Tree Interfaceの使用定義 (Copy 追加後)

```
1 typedef struct Tree<> {
2     union Data* tree;
3     struct Node* node;
4     __code put(Impl* tree, Type* node, __code next(...));
5     __code get(Impl* tree, Type* node, __code next(...));
6     __code remove(Impl* tree, Type* node, __code next(...));
7     __code copy(Impl* tree, __code next(...));
8     __code next(...);
9 } Tree;
```

10行目にcopy()が追加されており, inputDataGearは\_\_code nextのみとしている. これにより, goto tree->copy(next);といった記述でRedBlackTreeのコピーを行うことができる. 次にRedBlackTreeの実装の型定義をソースコード6.2に示す. 9, 12行目にある通り, コピー時に使用するtoStackを1つと木のコピーが完了しているかどうかを示すcopiedフラグを追加している. 今回の実装ではStackを2つ使用する. 追加したtoStackの他に, 元からRedBlackTreeの操作に使用しているnodeStackを用いる. これらのStackはdataとしてNode構造体のポインタを持つ.

ソースコード 6.2: RedBlackTreeの実装の型定義 (Copy 追加後)

```
1 typedef struct RedBlackTree <> impl Tree {
2     struct Node* root;
3     struct Node* current; // reading node of original tree;
```

```

4 | struct Node* previous; // parent of reading node of original tree;
5 | struct Node* newNode; // writing node of new tree;
6 | struct Node* parent;
7 | struct Node* grandparent;
8 | struct Stack* nodeStack;
9 | struct Stack* toStack;
10 | __code findNodeNext(...);
11 | int result;
12 | int copied;
13 | } RedBlackTree;

```

## 6.2 コピーのアルゴリズム

コピーは2つの Stack を使用し、left 方向から深さ優先で RedBlackTree のノードをリーフ方向に降りながら行う。図 6.1 にコピー時のある地点でのコピー元の木と2つの Stack の状態を例示する。leftDown を2回行い、tree->current が key が1のノードを指している。この時、木を辿るための nodeStack は tree->current 以前に辿った3と2のノードが積まれている。toStack にはコピー元の木のノードと key, value, color の値が同じでアドレスが異なるノードが積まれている。ノードは辿った際にコピーを行い toStack へと積むため、すでに辿った3, 2, 1のノードが積まれている。葉ノードまでたどり着いた時や left, right がすでに辿ったノードだった場合は up で root 方向に戻る。その際は、tree->current を親ノードに付け替え、nodeStack と toStack を1回 pop する。toStack は新規にアロケートした子ノードと親ノードを接続したり、すでにノードをアロケートしているかどうかを判断するなど、コピー先の木を操作したり状態を確認したりするために必要である。例えば、up する場合はコピー元の right ノードの存在を確認し、toStack の top をみてすでに right ノードを新規にアロケートしているかを見ることで、さらに up するかどうかを判断するなどがある。

図 6.2 に Copy 時の CodeGear の大まかな遷移を示す。四角ノードが CodeGear、丸ノードが遷移条件を表す。それらを囲む四角は CodeGear の遷移を LeftDown, RightDown, Up の3つのフェーズに区切ったもので、図を簡略化するためのものである。フェーズの四角に矢印が向くと、フェーズが持つ Start(オレンジで示される箇所)から遷移が始まる。青の Start, End は Copy の始めと終わりである。実際には leftDown などは leftDown1, leftDown2 のような複数の CodeGear で構成され、それぞれで Stack の操作やアロケート、ノードの存在確認を行う。Up はすでに木全体を辿ったかどうかを判定し、辿っていた場合は swap CodeGear へ遷移を行う。swap はコピー元の木とコピー先の木を入れ替える。これは Copying GC における From 領域と To 領域を入れ替える操作を想定している。

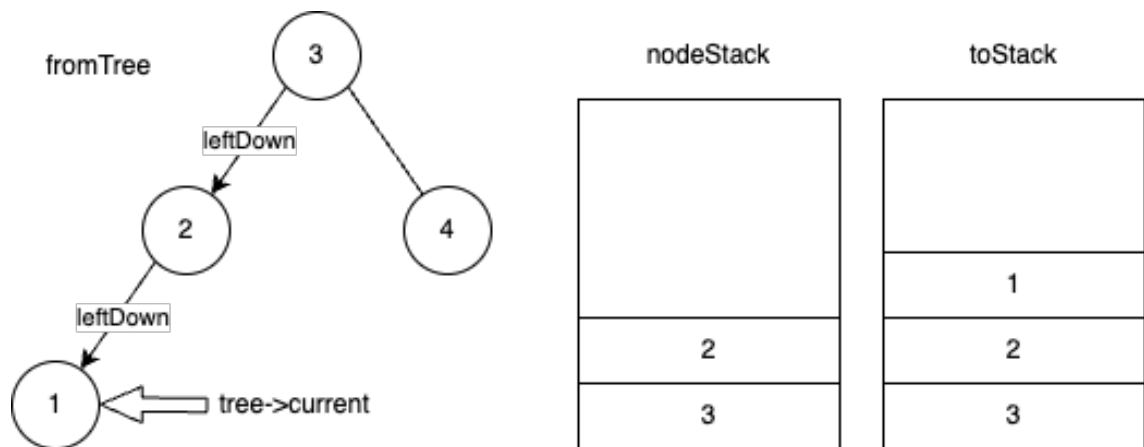


図 6.1: コピー元の Tree と 2 つの Stack の状態の例

### 6.3 コピーの実装の詳細

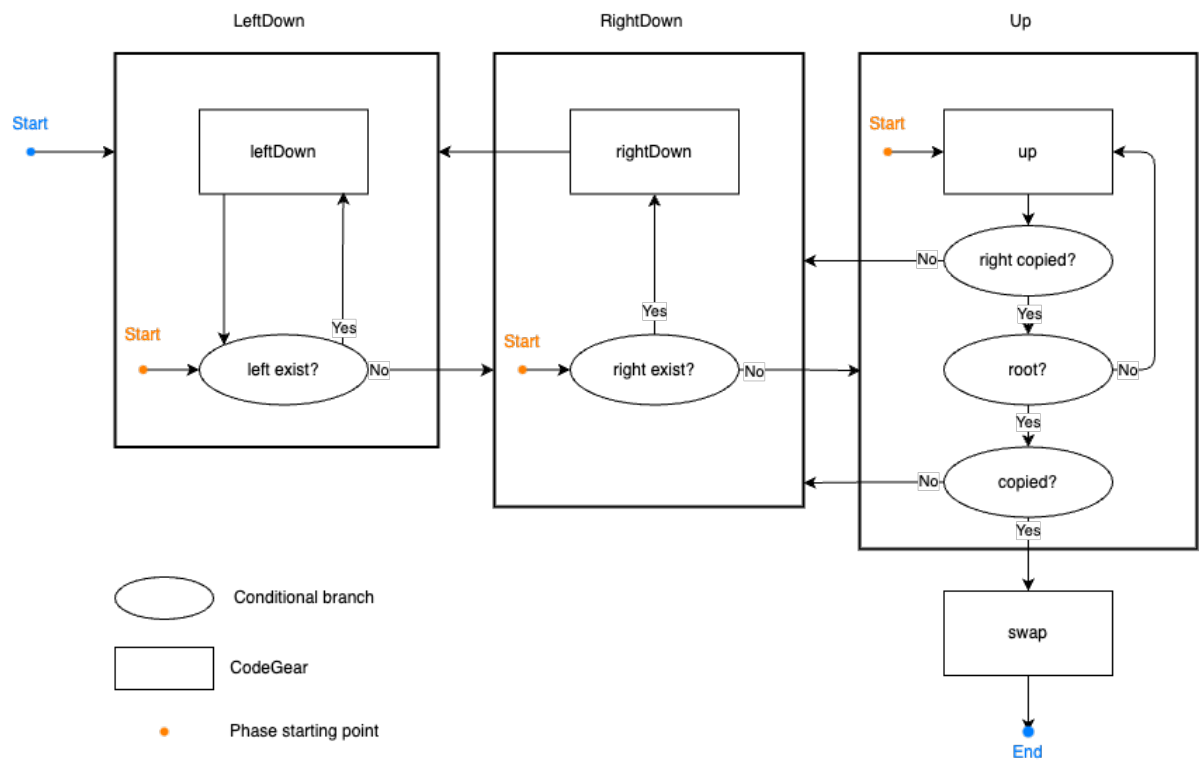


図 6.2: Copy 時の CodeGear の大まかな遷移

## 第7章 まとめと今後の課題

本研究では GearsOS のファイルシステムである GearsFileSystem における GC とレプリケーションについての考察, CopyRedBlackTree の実装と考察を行った.

また, 今後の課題や議題として次のようなものが挙げられる.

### 7.1 ファイルシステムにおけるスキーマ

従来の RDB のようなスキーマが存在すると, 個別にバックアップなどを取らない限りスキーマの変更以前にロールバックすることができない. しかしながら, 実際運用する上でスキーマを変更することは多々ある. これは, データの信頼性を低下させると考える. また, DB 上のデータ構造とプログラム上で扱うデータ構造に差が生まれるインピーダンスミスマッチが発生し, DB のデータをプログラムが扱う際にその差を埋めるような変換を必要とする場合が生まれる. 一方で, スキーマがあることによってデータに対して高度な操作を行うことができ, また, インデックスを容易に作成することができるといったメリットがある. よって, スキーマフルな DB とスキーマレスな DB はそれぞれメリットがあり, 状況によって使い分けるのが良いと考える.

今回は, 非構造化データ内であれば構造化データを扱うことが可能であることと, 信頼性を保証したいという点から, スキーマレスな DB としてのファイルシステムを考える. しかしながら, トランザクションの仕組みを作る上で RedBlackTree に対し, キーを設定することから完全なスキーマレスとは言えない構成となる.

GearsOS のデータは全て DataGear で表される. よって, GearsOS におけるファイルシステムは DataGear の集合となる. スキーマレスとはキーがない状態のことといえるが, DataGear はキーが設定されていないため, その集合自体にスキーマは無い. そのことから, GearsOS におけるスキーマとは DataGear 上のキーの構成であることがわかる. なお, 今回の RedBlackTree による構成の場合, キーは RedBlackTree を指す. DataGear は Kernel の Context からプロセスの Context を経由して全て繋がっている. よって, Kernel の Context 上にキーを用いた DataGear の参照を書き込む.

## 7.2 RedBlackTreeによる権限の表現

ファイルの権限はファイルシステムの重要な機能の一つであるため実装する必要がある。今回の RedBlackTree による構成の場合、木のルートの所持者を設定することがファイルに対して権限を設定することにあたる。ルートに対してアクセスする権限がなければ、読み書きができないといった実装になると考える。

## 7.3 データクエリ言語

ユーザーやアプリケーションがDBのデータにアクセスするための言語設計をする必要がある。RedBlackTreeのキーを用いたインデックスに対応し、従来のSQLと比較してより柔軟なクエリを実行できることを目指したい。

## 7.4 ログなどの時系列データの保存

時系列データは通常の方法と違った保存の方法を考えることができる。時系列に並んでいることを生かしたデータの保存方式や、時間経過に伴うデータの重要性の変化を考慮に入れたデータ圧縮の方法をとることで、より効率的な保存が可能だと考える。

## 7.5 スタンドアロンなDB

MySQLやPostgreSQLなどはOSの機能としてではなく、それ自体が一つのアプリケーションとして自立的に動作している。自立的に動作するのは、データのポータビリティを向上させるためである。スタンドアロンなDBの形にするか、その他の方法でポータビリティを向上させる手法を考えたい。

## 7.6 GCタイミングに必要な処理の自動挿入

# 謝辞

本研究を行うにあたりご多忙にも関わらず日頃より多くのご助言，ご指導をいただきました河野真治准教授に心より感謝いたします。また，学士時代に研究に対する意見，実装，実験に協力いただいた研究室OBの一木貴裕さんや，研究室の同期としてともに研究の遂行をしてくださった木山瑞基をはじめとする並列信頼研究室の皆さまに感謝いたします。最後に，長年に渡り理解を示し，支援してくださった家族に感謝いたします。

2024年3月  
又吉雄斗



## 参考文献

- [1] 一般社団法人全国銀行資金決済ネットワーク. 全国銀行データ通信システムの障害について. [https://www.zengin-net.jp/announcement/pdf/announcement\\_20231201.pdf](https://www.zengin-net.jp/announcement/pdf/announcement_20231201.pdf).
- [2] 全日本空輸株式会社. 4月3日に発生した国内線システム不具合の原因及び再発防止策について. <https://www.anahd.co.jp/group/pr/202304/notification-2.html>.
- [3] グローリー株式会社. 2月14日から2月19日にかけて発生した電子マネー決済システム (id 決済) の障害に関するお詫びとお知らせ. <https://www.glory.co.jp/news/detail/id=2017>.
- [4] 東恩納琢偉, 奥田光希, 河野真治 (琉球大学). Gears os でモデル検査を実現する手法について. 情報処理学会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 2020.
- [5] 一木貴裕. Gearsos の分散ファイルシステム設計. 修士 (工学) 学位論文, March 2022.
- [6] 又吉雄斗, 河野真治 (琉球大学). Gearsos における inode を用いたファイルシステムの構築, May 2022.
- [7] 並列信頼研究室. Cbc. <http://www.cr.ie.u-ryukyu.ac.jp/hg/CbC/CbC.llvm/>.
- [8] 河野真治. 継続を持つ c の下位言語によるシステム記述. 日本ソフトウェア科学会第 17 回大会論文集, September 2000.
- [9] 清水隆博. Gearsos のメタ計算. 修士 (工学) 学位論文, March 2021.
- [10] 並列信頼研究室. Gearsos. <http://www.cr.ie.u-ryukyu.ac.jp/hg/Gears/Gears/>.
- [11] 伊波立樹. Gearsos の並列処理. 修士 (工学) 学位論文, March 2018.
- [12] 並列信頼研究室. Gearsagda. <http://www.cr.ie.u-ryukyu.ac.jp/hg/Gears/GearsAgda/>.
- [13] 河野真治 (琉球大学) 森逸汰. Gearsagda による red black tree の検証, May 2023.

- [14] 並列信頼研究室. Cbc\_xv6. [http://www.cr.ie.u-ryukyu.ac.jp/hg/CbC/CbC\\_xv6/](http://www.cr.ie.u-ryukyu.ac.jp/hg/CbC/CbC_xv6/).
- [15] 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>.
- [16] 河野 真治 (琉球大学) 仲吉 菜々子. Gears os の codegear management, May 2023.
- [17] 河野 真治 (琉球大学) 一木 貴裕. Gearsos の分散ファイルシステムの設計. 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 2021.
- [18] 河野 真治 (琉球大学工学部情報工学科) 坂本 昂弘 (琉球大学工学部情報工学科). 継続を用いた xv6 kernel の書き換え. 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.
- [19] 清水隆博, 河野真治 (琉球大学). xv6 の構成要素の継続の分析. 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 2020.
- [20] 河野 真治. 分散フレームワーク christie と分散木構造データベース jungle, May 2018.

## 発表履歴

- 又吉 雄斗, 河野 真治. GearsOSにおけるinodeを用いたファイルシステムの構築. 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, 2022
- 又吉 雄斗, 佐野 巧曜, 河野 真治. Gears OS のファイルシステムとDB. 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, 2023

# 付録A 研究会業績

## A-1 研究会発表資料

- GearsOSにおけるinodeを用いたファイルシステムの構築 又吉 雄斗, 河野 真治 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, 2022
- Gears OSのファイルシステムとDB 又吉 雄斗, 佐野 巧曜, 河野 真治 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, 2023