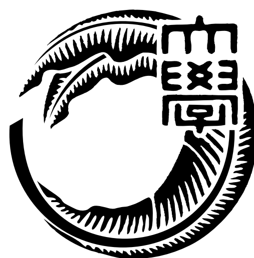
修士(工学)学位論文
Master's Thesis of Engineering

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 においてファイルシステムは重要な機能の一つであるため実装する必要がある．現在，一般的なアプリケーションはファイルシステムとデータベースを併用する形で構成される．アプリケーションのデータベースとしてファイルシステムを使用する構成が取れるようにしたい．ファイルシステムとデータベースの違いについて考え，データベースとしても利用可能なファイルシステムを構築したい．本研究では，ファイルシステムとデータベースの違いについて考察し，Gears OS のファイルシステムの設計について述べる．

Abstract

In our laboratory, we are developing GearsOS, aimed at ensuring reliability through the use of Continuation based C (CbC) along with theorem proving and model checking. Implementing a file system is a necessary task in an OS as it's one of the critical features. Currently, general applications are structured to use both file systems and databases. While databases allow data insertion and modification through SQL, they require predefined schemas and the specification of these schemas at the time of insertion. In the GearsOS file system, we aim to implement interfaces equivalent to SQL functions such as grep and find, enabling the use of the file system as a database for applications. We want to construct a file system that can also function as a database by considering the differences between file systems and databases. This research discusses these differences and describes the design of the file system for Gears OS.

目次

第 1 章　GearsOS におけるファイルシステムと DB　5

第 2 章　軽量継続を基本とする言語 CbC　7

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

第 3 章　信頼性の保証を目的とした GearsOS　10

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 14

第 4 章　GearsOS のファイルシステム　15

4.1 DataGearManager による分散ファイルシステム 15
4.2 i-node を用いたファイルシステム 15
4.3 非破壊 RedBlackTree による構成 15
4.4 RedBlackTree のトランザクション 15
4.5 ディスク上とメモリ上のデータ構造 16

第 5 章　GearsFileSystem における GC とレプリケーション　20

5.1 ファイルシステムの信頼性に関する機能 20
5.2 メモリの管理手法 20
5.3 GearsFileSystem の GC 21
5.4 GearsFileSystem のレプリケーション 22

第 6 章　CopyRedBlackTree の実装　24

6.1 コピーのアルゴリズム 26
6.2 Stack の使用に関して 27
6.3 証明のしやすさについて 27

第 7 章　まとめと今後の課題 **28**
7.1 ファイルシステムにおけるスキーマ 28
7.2 RedBlackTree による権限の表現 29
7.3 データクエリ言語 29
7.4 ログなどの時系列データの保存 29
7.5 スタンドアロンな DB 29

謝辞 **30**

参考文献 **31**

研究関連論文業績 **33**

付録 **33**

付 録 A 研究会業績 **34**
A-1 研究会発表資料 34

図目次

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

4.1 トランザクショナルな write 操作 18
4.2 非破壊的な Tree 編集 19

5.1 RedBlackTRee の Copy による GC 22
5.2 Copy のアルゴリズム 23

6.1 Copy のアルゴリズム 27

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
6.1 CopyRedBlackTree の実装 24
6.2 CopyRedBlackTree のアルゴリズム 26

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などに必要不可欠であるため，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]．

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，データを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る．

データのレプリケーションやガベージコレクションの仕組みに必要である, RedBlackTree の Copy の仕組みを実装した.

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．これを通常の関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，通常の関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．通常の関数の場合，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

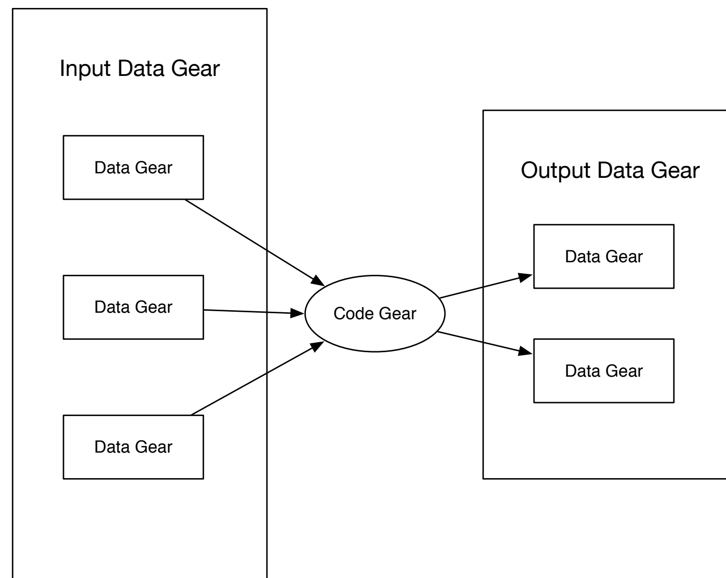


図 2.1: CodeGear と入出力の関係図

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 *arcv[]) {
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種類の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．

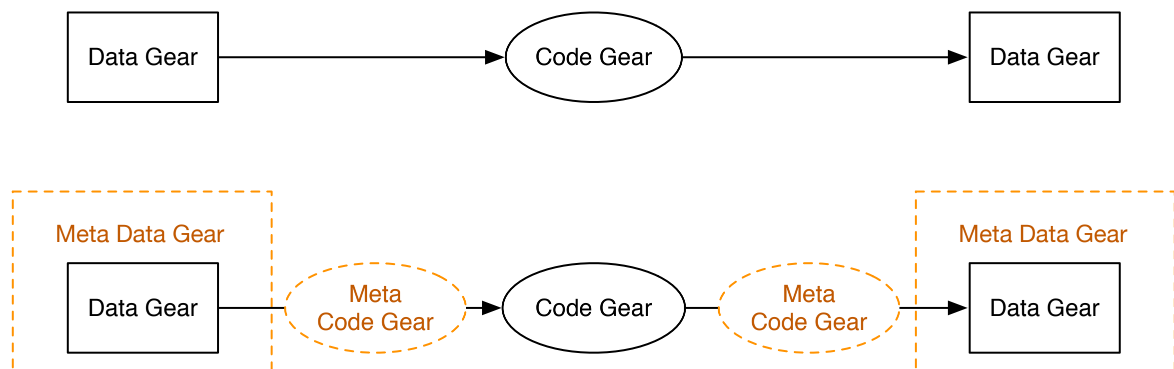


図 3.1: CodeGear と MetaCodeGear の関係

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 へデータを 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 へデータを 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.

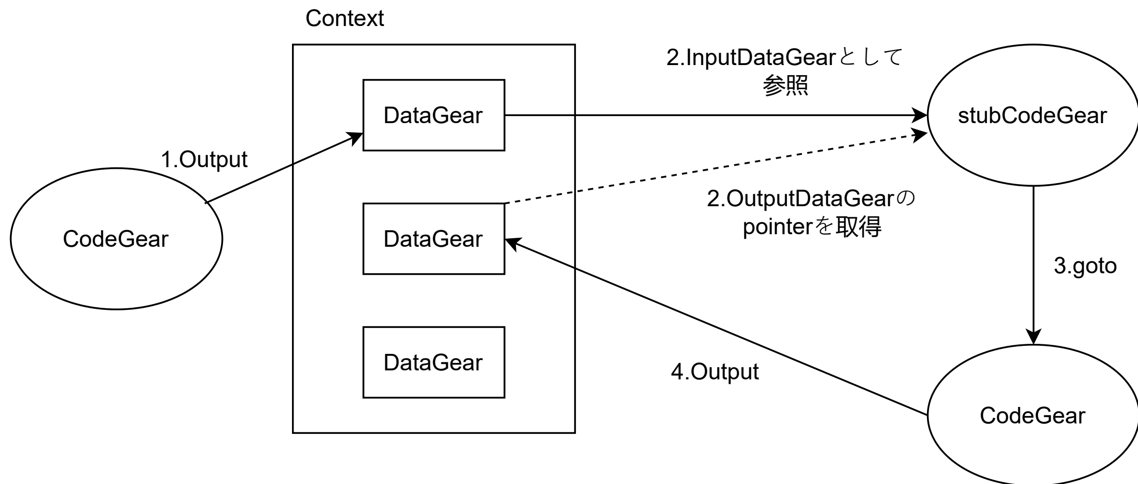


図 3.2: Context を参照する流れ

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，data，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．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 の実装似ついて説明する．Queue interface の実装の一つである SingleLinkedQueue 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 "SingleLinkedQueue.h"
4  #data "Node.h"
5  #data "Element.h"
6
7  Queue* createSingleLinkedQueue(struct Context* context) {
8      struct Queue* queue = new Queue();
9      struct SingleLinkedQueue* singleLinkedQueue = new SingleLinkedQueue()
      ;
10     queue->queue = (union Data*)singleLinkedQueue;
11     queue->take  = C_takeSingleLinkedQueue;
12     queue->put  = C_putSingleLinkedQueue;
13     queue->isEmpty = C_isEmptySingleLinkedQueue;
14     queue->clear = C_clearSingleLinkedQueue;
```

```
15     singleLinkedQueue->top  = new Element();
16     singleLinkedQueue->last = singleLinkedQueue->top;
17     return queue;
18 }
19
20 //  省略~~~~
21
22 __code takeSingleLinkedQueue(struct SingleLinkedQueue* queue, __code next
       (union Data* data, ...)) {
23     printf("take\n");
24     struct Element* top = queue->top;
25     struct Element* nextElement = top->next;
26     if (queue->top == queue->last) {
27         data = NULL;
28     } else {
29         queue->top = nextElement;
30         data = nextElement->data;
31     }
32     goto next(data, ...);
33 }
34
35 // 省略~~~~
```

3.5 GearsOS の RedBlackTree

第4章　GearsOSのファイルシステム

4.1　DataGearManagerによる分散ファイルシステム

4.2　i-nodeを用いたファイルシステム

4.3　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．

4.4　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 4.1 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．

4.5 ディスク上とメモリ上のデータ構造

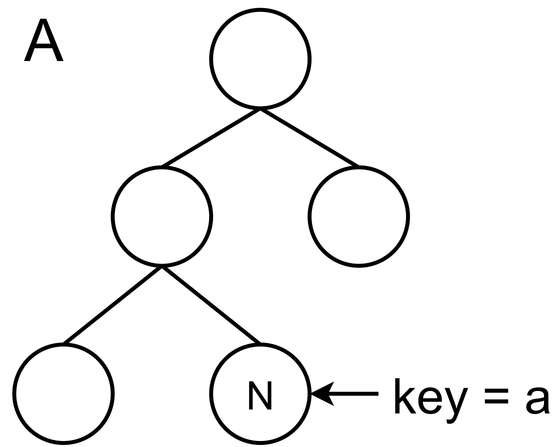
ファイルシステムは全て RedBlackTree で構成する．それにより，プログラムの証明がより簡単になるからである．また，ファイルシステムと DB はデータを保管するという本質的な役割は同じである．よって，それらはまとめて RedBlackTree で構成する．

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 4.2 は非破壊的編集を行う RedBlack-Tree 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.



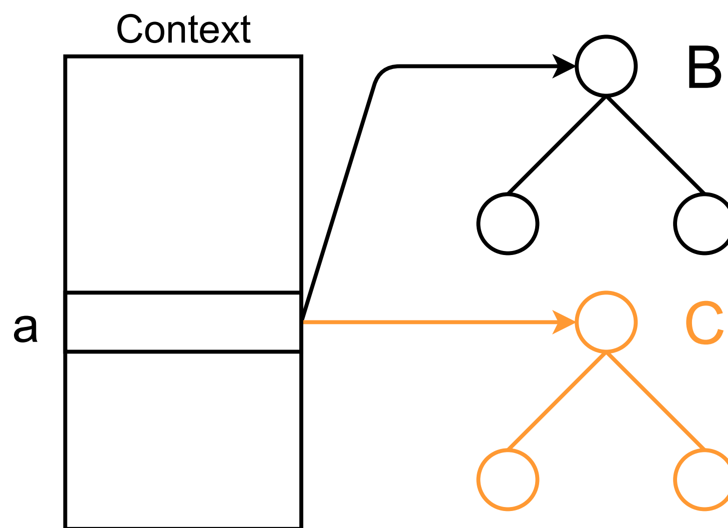


図 4.1: トランザクショナルな write 操作

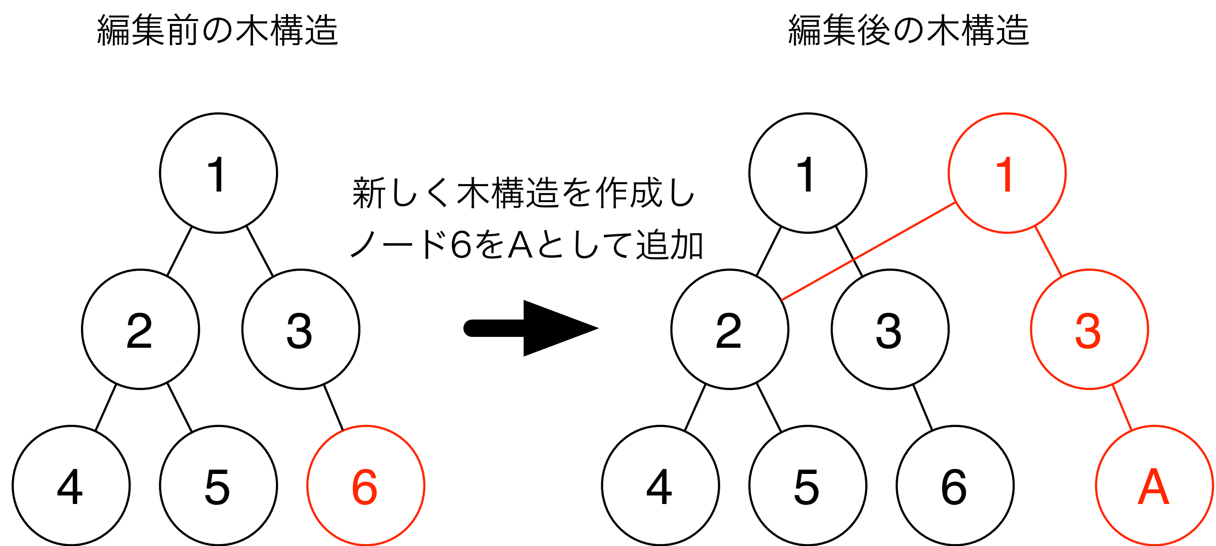


図 4.2: 非破壊的な Tree 編集

第5章　GearsFileSystemにおけるGCとレプリケーション

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．これらの機能を実装することでファイルシステムの信頼性を高めたい．

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，カウンターが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，カウンター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．

今回は，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構築することを考える．非破壊的な Tree 編

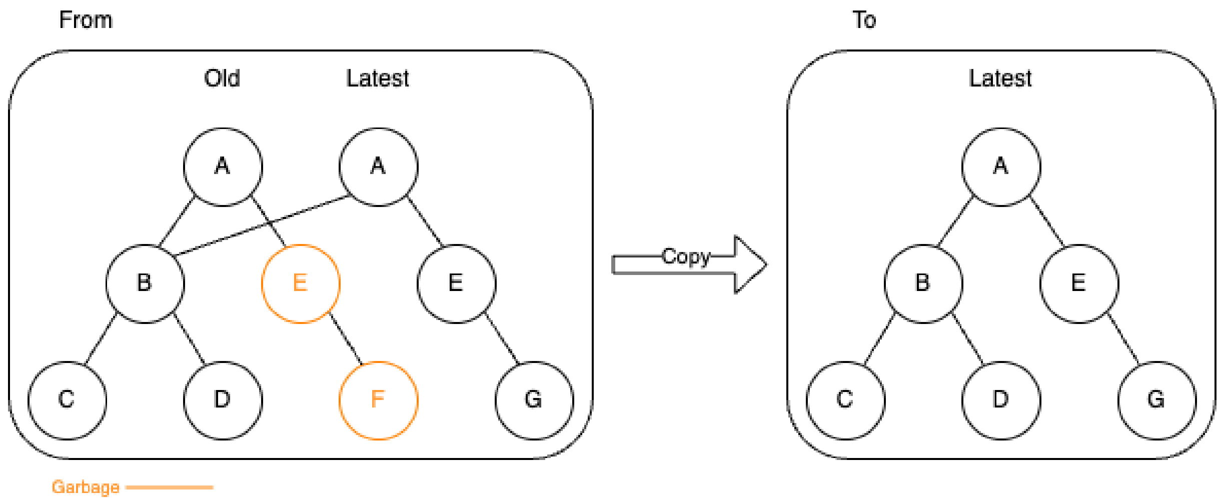


図 5.1: RedBlackTRee の Copy による GC

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

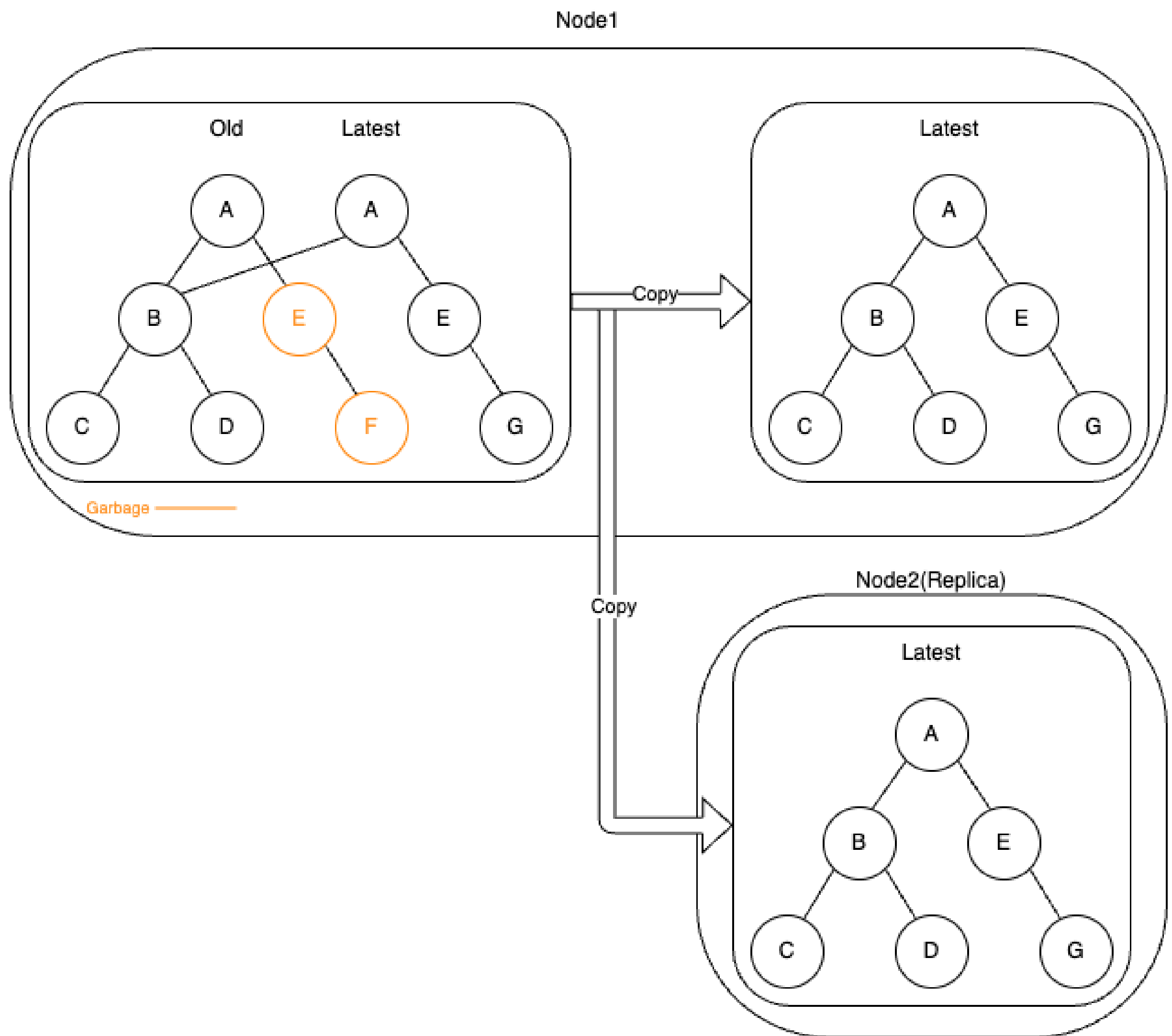


図 5.2: Copy のアルゴリズム

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．

ソースコード 6.1: CopyRedBlackTree の実装

```
1  __code copyRedBlackTree(struct RedBlackTree* tree) {
2      tree->current = tree->root;
3      struct Stack* nodeStack = tree->nodeStack;
4      struct Node* oldNode = tree->current;
5      struct Node* newNode = tree->newNode;
6      tree->previous = newNode;
7      newNode = oldNode;
8      goto nodeStack->push((union Data*)newNode, leftDown);
9  }
10
11 __code leftDown(struct RedBlackTree* tree, struct Stack* inputStack,
       struct Stack* outputStack) {
12     struct Node* newNode = &ALLOCATE(context, Node)->Node;
13     newNode->key = tree->current->key;
14     // newNode->value = tree->current->value;
15     newNode->value = (union Data*)new Integer();
16     ((Integer*)newNode->value)->value = ((Integer*)tree->current->value)
       ->value;
17     newNode->color = tree->current->color;
18     newNode->right = tree->current->right;
19     struct Stack* nodeStack = tree->nodeStack;
20     printf("leftDown\n");
21     goto nodeStack->push(newNode, leftDown1);
22 }
23
24 __code leftDown1(struct RedBlackTree* tree, struct Stack* inputStack,
       struct Stack* outputStack) {
25     struct Stack* nodeStack = tree->nodeStack;
26     printf("leftDown1\n");
27     if (tree->current->left) {
28         goto leftDown(tree->current->left, inputStack, outputStack);
29     } else if (tree->current->right) {
30         goto rightDown(tree->current->right, inputStack, outputStack);
31     } else {
32         goto nodeStack->pop(up);
```

```
33     }
34 }
35 
36 __code rightDown(struct RedBlackTree* tree, struct Stack* inputStack,
     struct Stack* outputStack) {
37     struct Node* newNode = &ALLOCATE(context, Node)->Node;
38     newNode->key = tree->current->key;
39     newNode->value = (union Data*)new Integer();
40     ((Integer*)newNode->value)->value = ((Integer*)tree->current->value)
     ->value;
41     newNode->color = tree->current->color;
42     newNode->right = tree->current->right;
43     tree->current = tree->current->right;
44     struct Stack* nodeStack = tree->nodeStack;
45     printf("rightDown\n");
46     goto nodeStack->push(newNode, leftDown);
47 }
48 
49 __code up(struct Node* node, struct RedBlackTree* tree, struct Stack*
     inputStack, struct Stack* outputStack) {
50     struct Stack* nodeStack = tree->nodeStack;
51     struct Node* newNode = tree->newNode;
52     printf("up\n");
53     if (node->left) {
54         tree->current = node->right;
55         node->left = newNode;
56         goto nodeStack->push((union Data*)node, rightDown);
57     } else {
58         node->right = newNode;
59         newNode = node;
60         goto nodeStack->isEmpty(popWhenNoEmpty, popWhenEmpty);
61     }
62 }
63 
64 // するときにの情報を元にを付け替えないといけないのでは？ upstackcurrent
65 __code up1(struct Node* node, struct RedBlackTree* tree, struct Stack*
     inputStack, struct Stack* outputStack) {
66     struct Stack* nodeStack = tree->nodeStack;
67     printf("up1\n");
68     goto nodeStack->pop(up2);
69 }
70 
71 __code up2(struct Node* node, struct RedBlackTree* tree, struct Stack*
     inputStack, struct Stack* outputStack) {
72     printf("up2\n");
73     tree->current->right = node->right;
74     goto up(tree, inputStack, outputStack);
75 }
76 
77 __code popWhenEmpty(struct Node* node, struct RedBlackTree* tree, struct
     Stack* inputStack, struct Stack* outputStack, __code next(...)) {
78     // の入れ替え tree
79     struct Tree* newTree = new Tree();
```

```
80      struct RedBlackTree* newRedBlackTree = new RedBlackTree();
81      newTree->tree = (union Data*)newRedBlackTree;
82      newRedBlackTree->root = tree->newNode;
83      newRedBlackTree->current = tree->newNode;
84      tree = newRedBlackTree;
85      printf("popWhenEmpty\n");
86      goto next(...);
87  }
88
89  __code popWhenNoEmpty(struct Node* node, struct RedBlackTree* tree,
        struct Stack* inputStack, struct Stack* outputStack) {
90      struct Stack* nodeStack = tree->nodeStack;
91      printf("popWhenNoEmpty\n");
92      goto nodeStack->pop(up1);
93  }
```

6.1 コピーのアルゴリズム

ソースコード 6.2: CopyRedBlackTree のアルゴリズム

```
1   __code leftDown(tree, inputStack, outputStack) {
2       // ノードをアロケートしてに入れる inputStack
3       // if tree->left goto leftDown(tree->left, inputStack, outputStack)
4       // else if tree->right goto rightDown(tree->right, inputStack,
        outputStack)
5       // else goto up(tree, inputStack, outputStack)
6   }
7   __code rightDown(tree, inputStack, outputStack) {
8       // ノードをアロケートしてに入れる inputStack
9       // 新しく作ったノードをに入れる leftoutputStack
10      // if tree->left goto leftDown(tree->left, inputStack, outputStack)
11      // else if tree->right goto rightDown(tree->right, inputStack,
        outputStack)
12      // else goto up(tree, inputStack, outputStack)
13  }
14  __code up(tree, inputStack, outputStack) {
15      // if 左からきたら inputStack->top->を埋める left
16      //    goto rightDown()
17      // if 右からきたら
18      //     pop inputStack
19      //     pop outputStack
20      //     inputStack->top->を埋めて
        rightgoto up(tree, inputStack, outputStack)
21  }
```

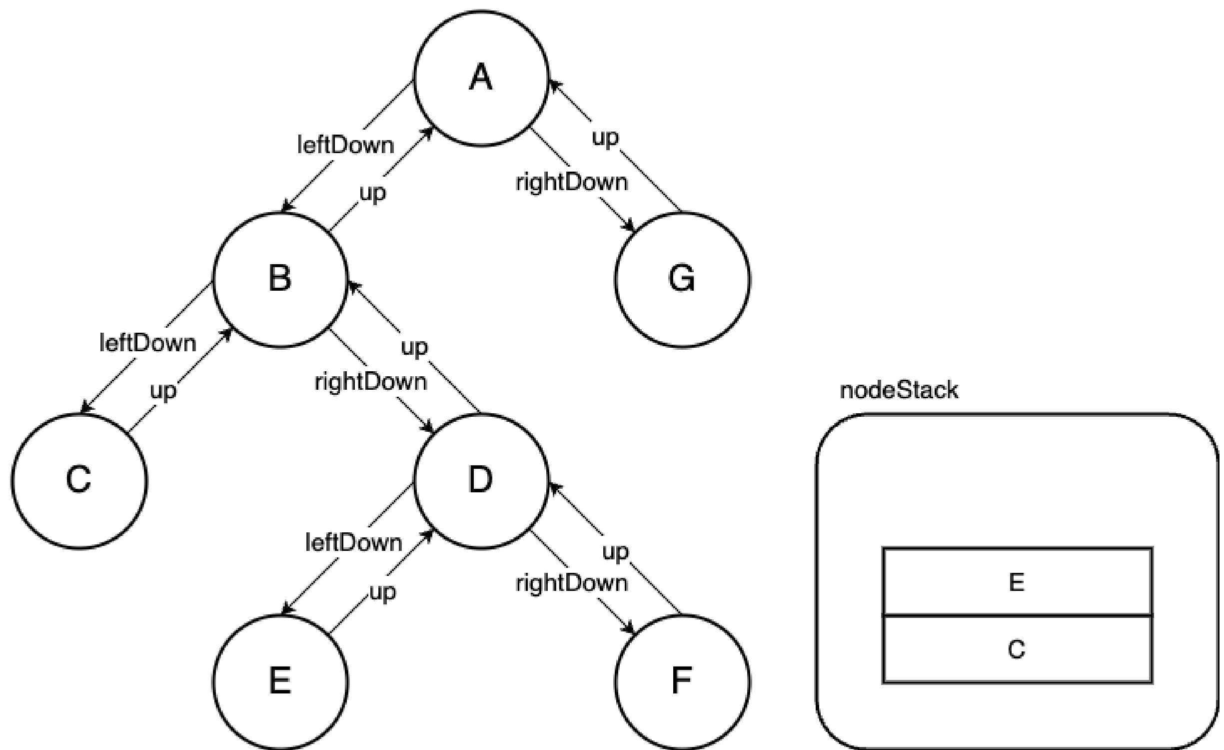


図 6.1: Copy のアルゴリズム

6.2 Stack の使用に関して

6.3 証明のしやすさについて

第7章　まとめと今後の課題

本論文では GearsOS のファイルシステムと DB の設計について説明した．今後，実装を行いながら設計と動作の確認，計測を行い，設計の意図が反映されていることやプログラムの検証ができることを確認する必要がある．

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デメリットがあり，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データ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．

謝辞

本研究の遂行、本論文の執筆にあたり、

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.

[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/.

[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