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

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を行なってきた．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，非破壊構造の増大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際は，データをコピーする機能が必要不可欠である．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ものの，現状は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，よりシンプルかつデータの持続性が確保された，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 laboratory, we are developing GearsOS with the aim of ensuring reliability through theorem proving and model checking, utilizing Continuation based C (CbC). One of the critical components of an operating system is its file system. In GearsOS, we have implemented mechanisms for distributed file systems as well as i-node based file system architectures. However, the current file system in GearsOS lacks replication and backup functionalities for data multiplicity assurance, as well as garbage collection features to address non-destructive structure growth and memory fragmentation. Implementing these features at the file system level in GearsOS would enhance data redundancy and memory safety significantly. Implementing garbage collection and replication is imperative, necessitating the functionality to copy data. In GearsOS, data is stored in the format of RedBlackTrees. While copying data can be achieved by duplicating RedBlackTrees, currently, there is no functionality within GearsOS's RedBlackTrees for tree duplication. This research focuses on designing, constructing, and discussing the necessary copy functionality for RedBlackTrees to implement file system replication and garbage collection features. By unifying these functionalities through RedBlackTree copying, our goal is to realize a simpler system with ensured data persistence, facilitating the application of formal methods such as theorem proving.

目 次

図 目 次

ソースコード目次

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，航空機予約，電子決済などのシステムで障害が発生し，社会的な影響を及ぼした [1, 2, 3]．これらのシステムの不具合を防ぎ，提供者や利用者のリスクを最小限に抑えるためには，堅牢なシステム構築が不可欠である．アプリケーション，オペレーティングシステム，データベース，そして物理的なコンポーネントまで，様々な要素が連携してこれらのシステムを支えており，これらの一つ一つに対する厳密な検証が，全体としての堅牢性を高めることに繋がる．

当研究室では，信頼性の保証を目的とした GearsOS を開発しており，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]．一般的にソフトウェアの動作を検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．よって，GearsOS はテストコードに加え，形式手法を用いることでより厳密に動作検証を行うことができるソフトウェアの構築を目指す．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，形式手法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．

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をしてきた [5, 6]．それらのファイルシステムは基本構造として非破壊 RedBlackTree を持つ．しかし，非破壊 RedBlackTree はデータが無尽蔵に増加するため，実用上の問題があると言える．よって，データの増加を防ぐような仕組みが必要である．また，本システムには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 のデータの多重度や一貫性の確保，非破壊 RedBlackTree の無尽蔵な増加を防ぐための GC とレプリケーション，バックアップ機構の設計を行い，それらを実現するために必要な RedBlackTree のコピーの仕組みの設計，構築，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

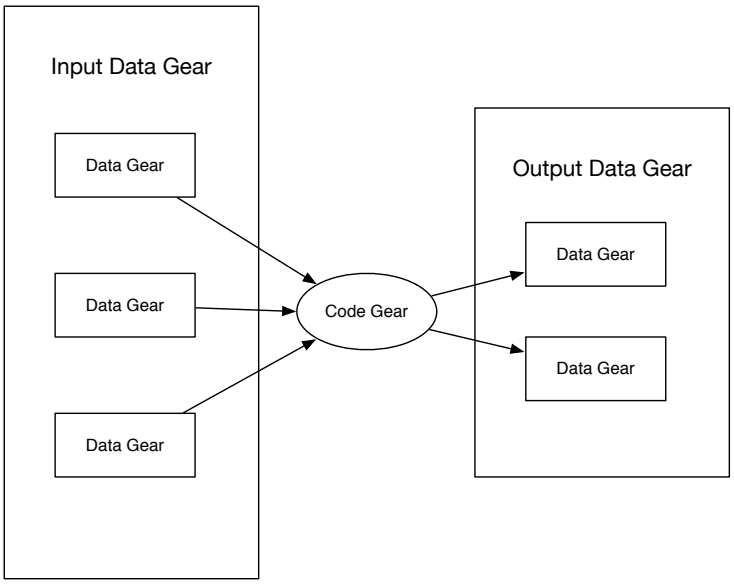

図 2.1: CodeGear と入出力の関係図

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 [9]．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，最後は exit_code へ goto することで処理を終了する．

ソースコード 2.1: CbC のプログラム例

```
__code add1(int in_n) {
    int out_n = n + 1;
    goto add2(out_n);
}

__code add2(int in_n) {
    int out_n = n + 2;
    goto end(out_n);
}

__code end(int in_n) {
    printf("%d", n);
    goto exit_code();
}

int main(int argc, char *arcv[]) {
    int n = 1;
    goto add1(n);
}
```

第3章　信頼性の保証を目的とした GearsOS

GearsOS[10, 11, 12] 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 [13]．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 [14]．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 [15]．これは，教育用に開発された x.v6[16] 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 [17]．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．

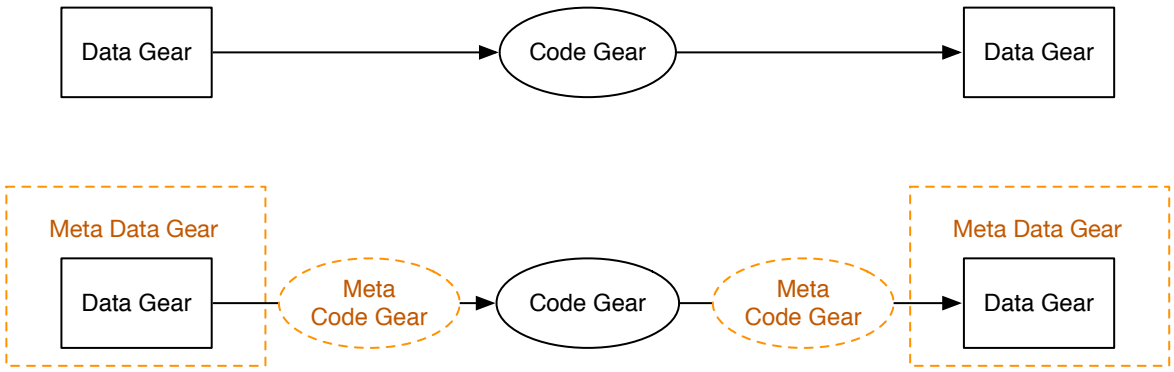

図 3.1: CodeGear と MetaCodeGear の関係

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．

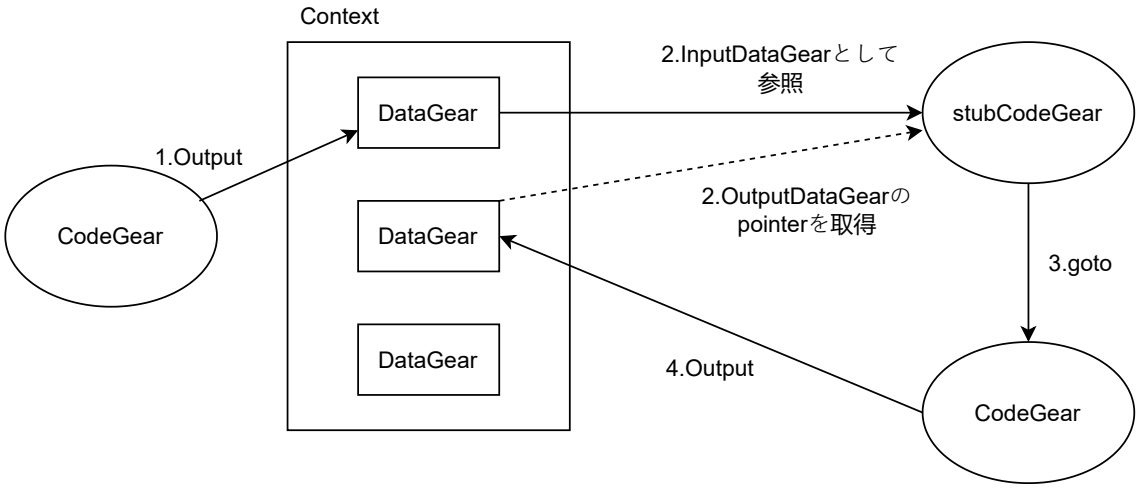

図 3.2: Context を参照する流れ

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 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 putSingleLinkedQueue(struct SingleLinkedQueue* 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 takeSingleLinkedQueue(struct SingleLinkedQueue* 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．そのため，`#impl "Queue.h" as "SingleLinkedQueue.h"`は仕様 Queue の実装を SingleLinkedQueue として定義することになる．7 行目の `createSingleLinkedQueue` は SingleLinkedQueue 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 SingleLinkedQueue 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 `putSingleLinkedQueue` や `takeSingleLinkedQueue` 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: SingleLinkedQueue の型定義

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

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 SingleLinkedQueue 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

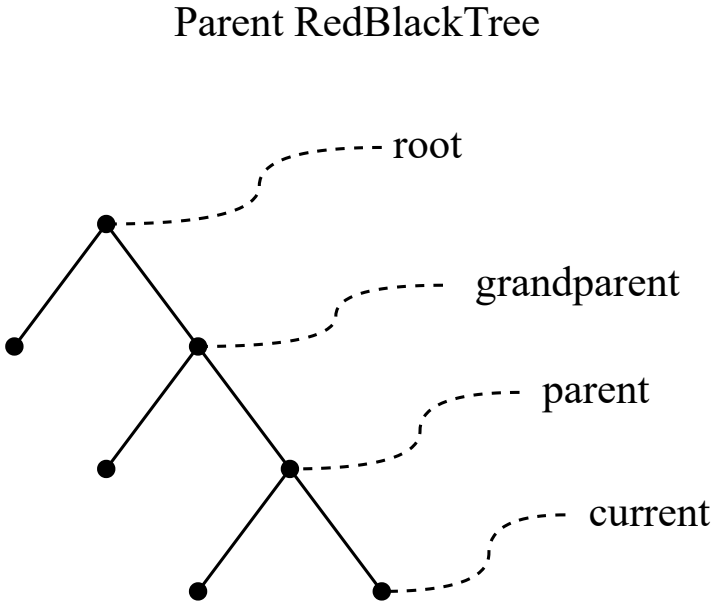

図 3.3: RedBlackTree のノードの種類

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．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

DataGear を用意する際は図 3.3 の 15 行目にあるように，`singleLinkedQueue->top = new Element();` のような形で new キーワードを用いる．これはビルド時に生成される ALLOCATE マクロに変換される．図 3.9 は ALLOCATE マクロの例である．ALLOCATE マクロは Context(context) と用意したい DataGear の型名 t を渡す．context は `context->heap` で示されるヒープ領域を持っており，この領域に DataGear を保持する．6 行目で DataGear のサイズ分のメモリ領域をヒープ上に確保していることがわかる．なお，ALLOCATE マクロを直接呼ばずに new キーワードで記述するのは，ノーマルレベルから metaDataGear に当たる Context を直接参照しないようにするためである．

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) ,
       sizeof(void *)) - sizeof(Meta); \
3      Meta* meta = (Meta*)context->heap;\
4      context->heap += sizeof(Meta);\
5      union Data* data = context->heap; \
6      context->heap += sizeof(t); \
7      meta->type = D_##t; \
8      meta->size = sizeof(t);     \
9      meta->len = 1;\
10     meta->data = data; \
11     *context->metaData = meta; \
12     context->metaData++; \
13     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の取り組みを行ってきた．

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 -i`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 -i`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．

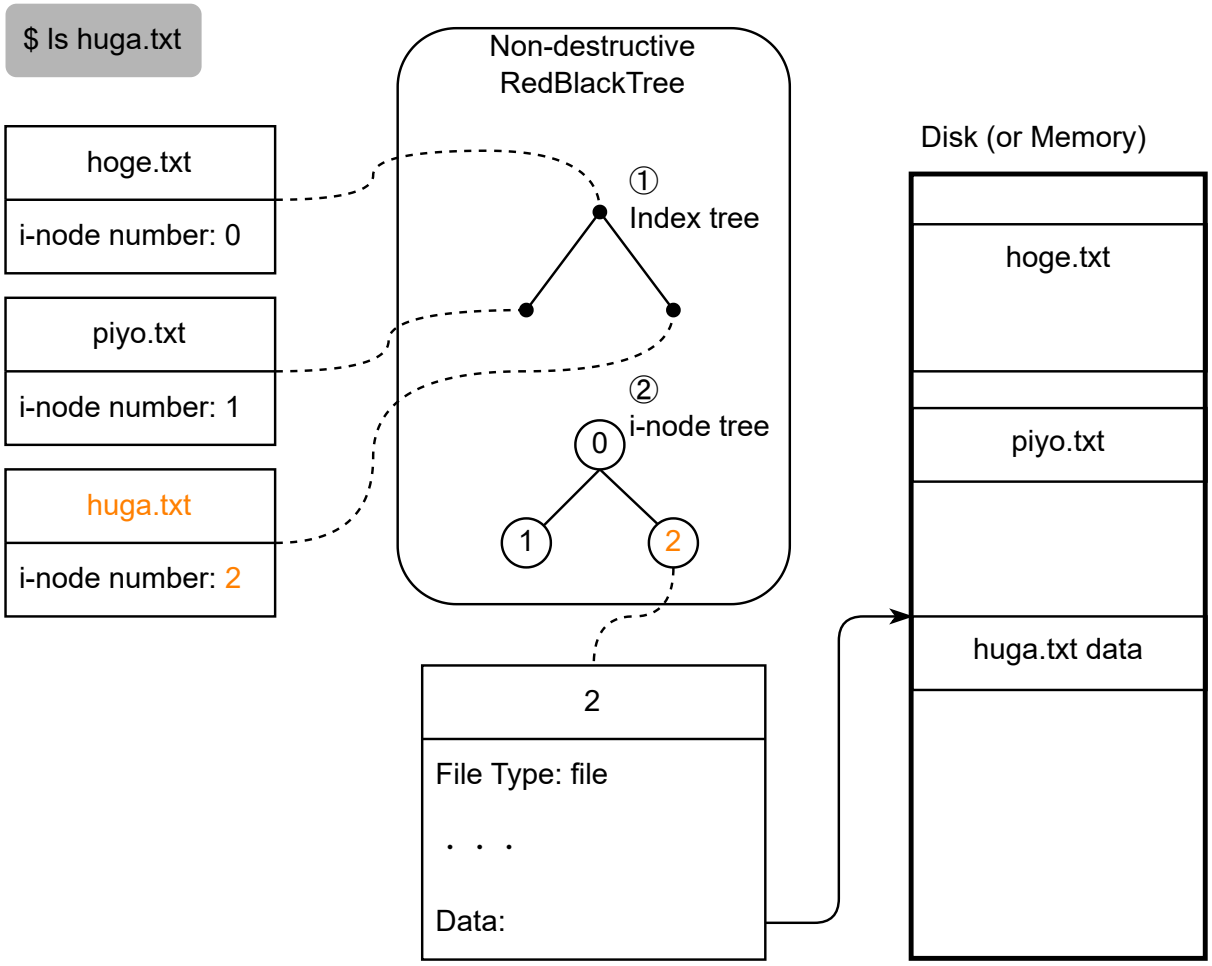

図 4.1: i-node を用いたディレクトリシステムの処理の流れ

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．

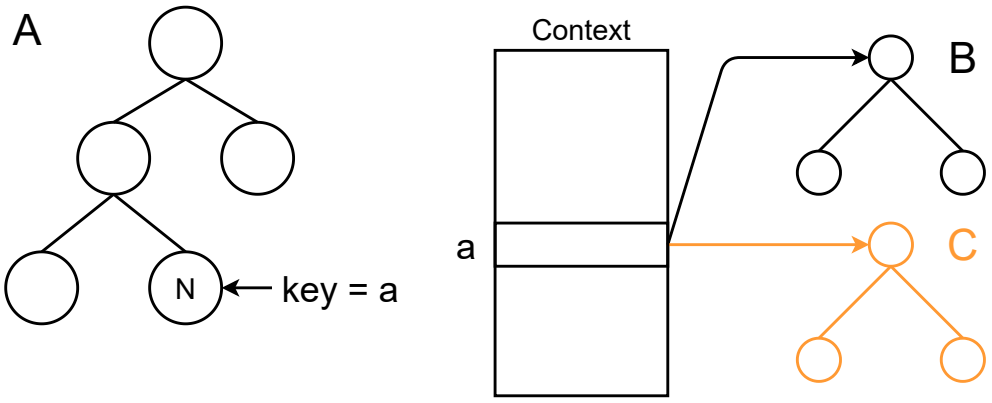

図 4.2: トランザクショナルな write 操作

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．

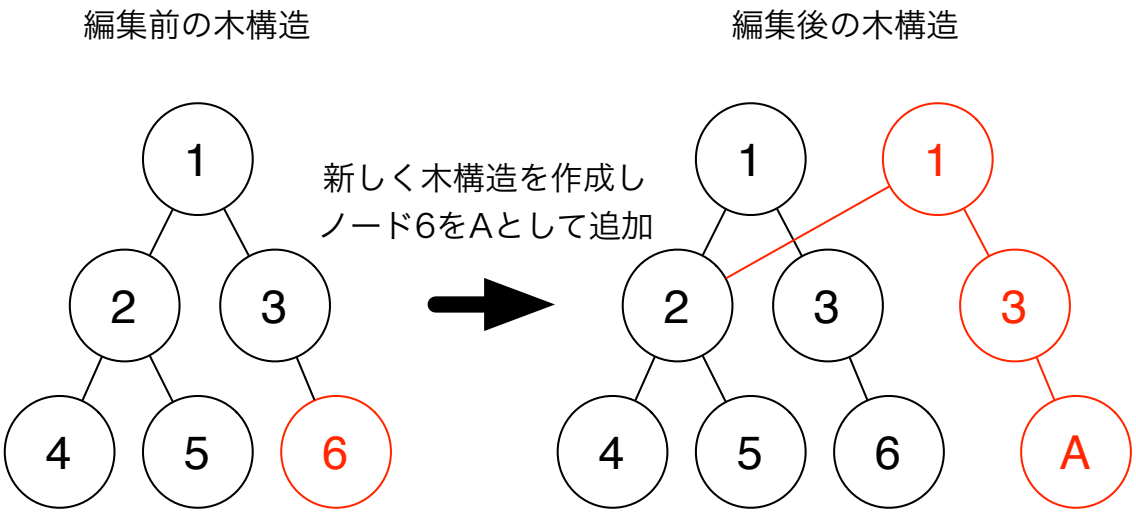

図 4.3: 非破壊的な Tree 編集

4.5 DataGearManager による分散ファイルシステム

本研究室では, 一木 貴裕 [5]GearsOS の分散ファイルシステム設計が行われ, GearsOS の分散ファイルシステムに必要な DataGearManager の通信路の実装がされた．DataGearManager(以下 DGM とする) はノード間で DataGear のやり取りをするための通信路を構成するための仕組みである．図 4.4 に DGM の Socket 通信による WordCount 例題の動作を示す．DGM は RemoteDGM と LocalDGM がペアとなり，RemoteDGM が送信先ノードの LocalDGM に接続することで DataGear の送信を可能にする．Remote から Local への片側方向通信となるため，相互通信したい際は 2 ペアで構成する．この際の接続は C 言語の Socket インターフェースを使用している．Node B の Local DGM が受け取ったファイルのデータを WordCount の CodeGear が処理をし，Node A の LocalDGM に接続された RemoteDGM に処理結果を Put することで，Node A に処理結果を返している．このように，別 Node からデータを受け取り，そのデータをしょりして受け取り元に処理結果などを返すことが可能な仕組みである．

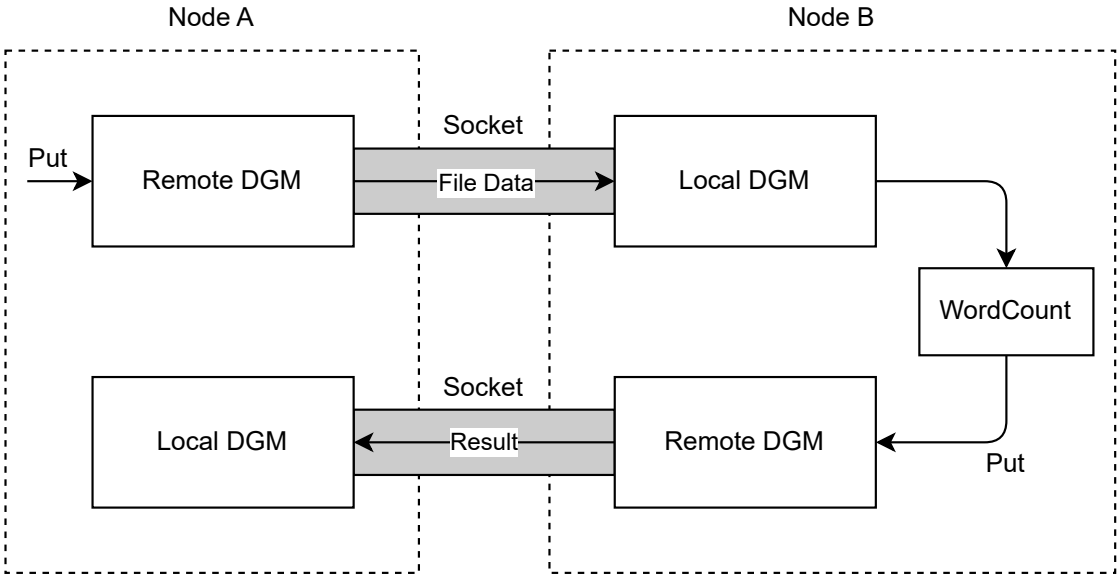

図 4.4: DGM の Socket 通信による WordCount 例題

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，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．これらの機能をファイルシステムのレベルで実装することで，より堅牢な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，データの持続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

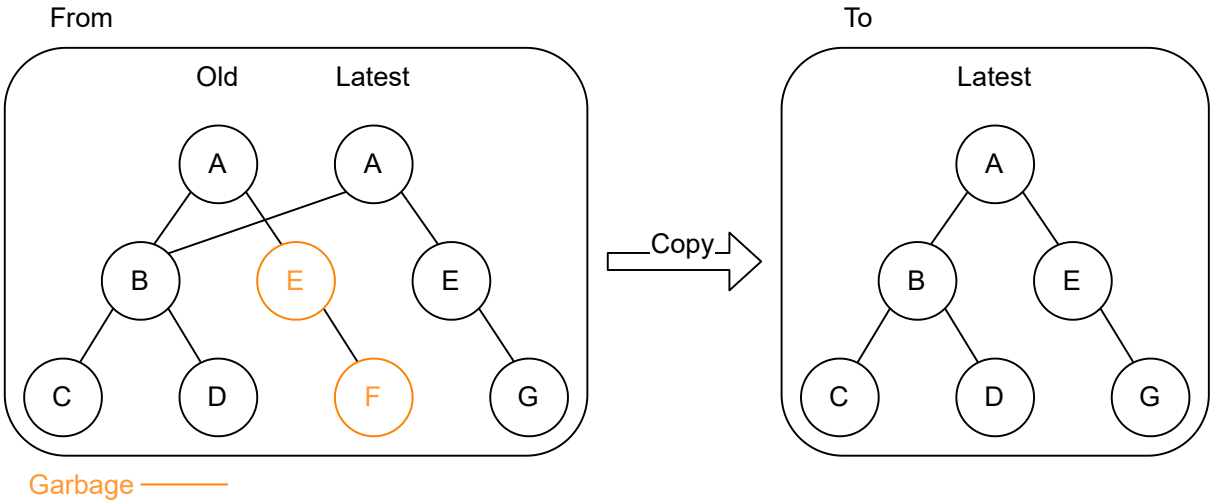

図 5.1: RedBlackTRee の Copy による GC

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．

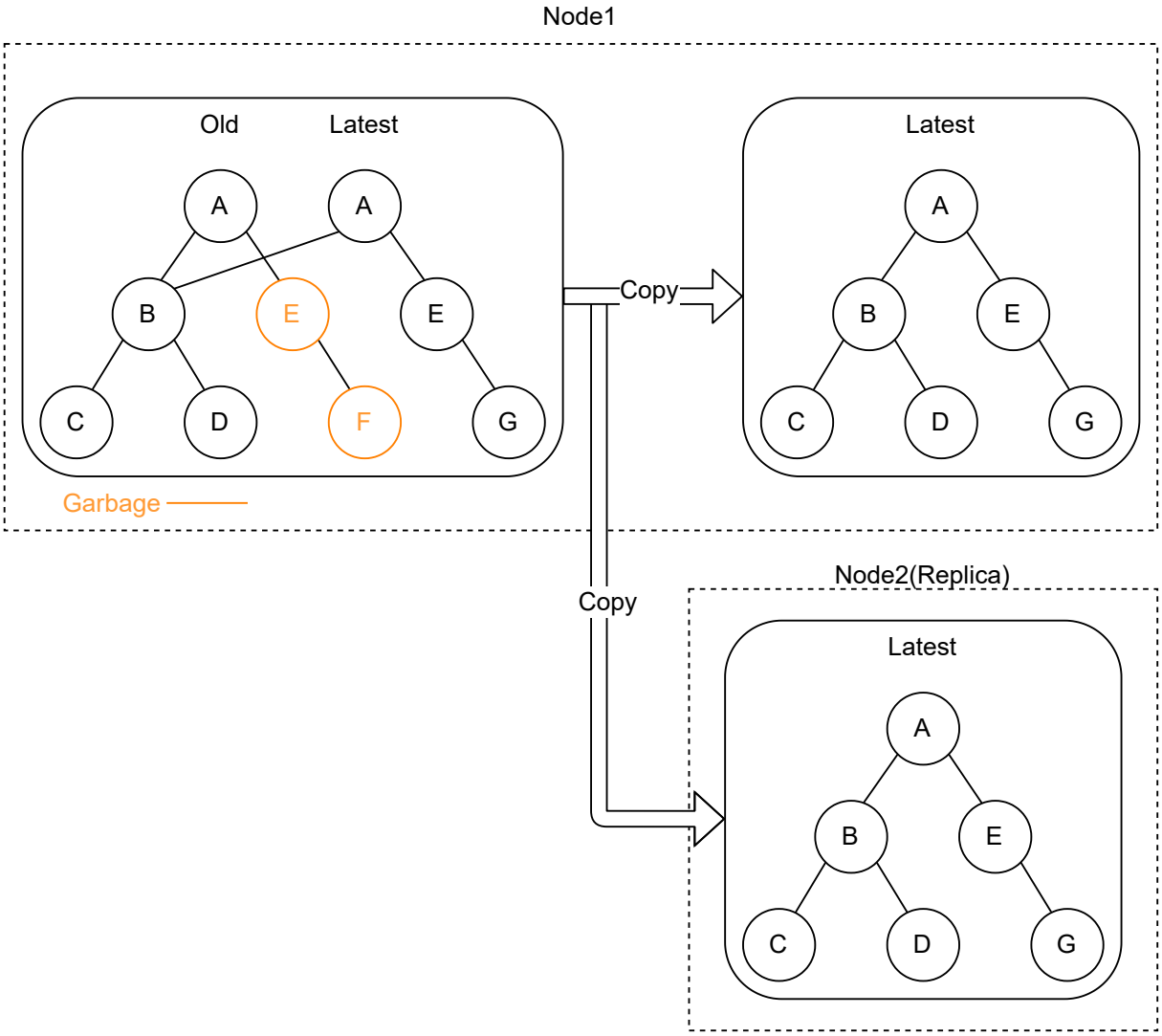

図 5.2: GearsFileSystem のレプリケーションの考え方

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の通信の仕組みについて考える必要がある．データの送受信は DataGearManager の Socket 通信の仕組みを使うことが考えられる．

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

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．

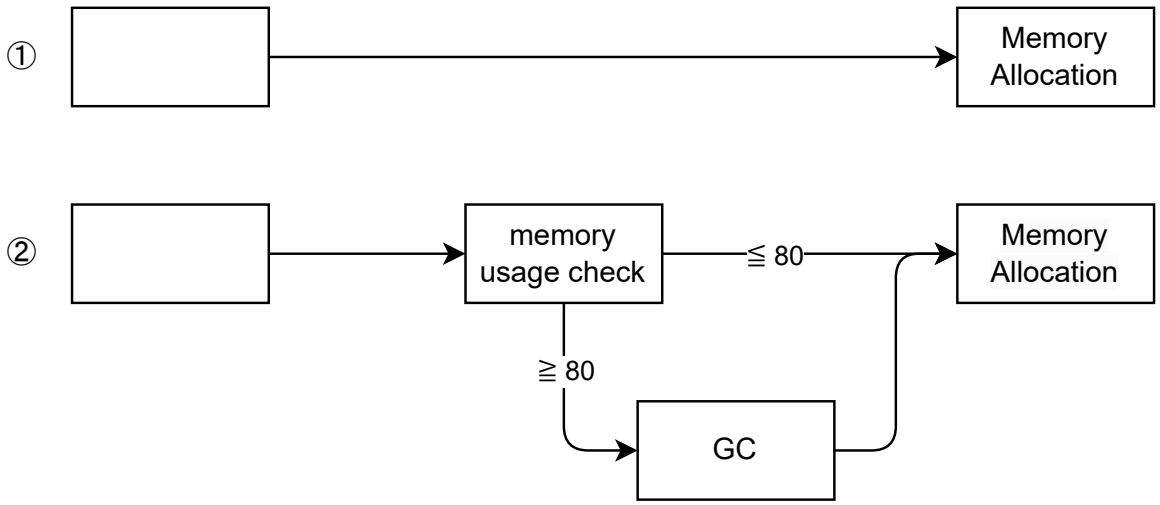

図 5.3: GC 実行処理の挿入

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．これは，GC処理を自動的に

5.6　別Contextへのコピー

図5.4にContextとCode Table，Data Tableの関係と，の複数のContextがある状態を示す．Contextはそれぞれ Code TableとData Tableを持つ．DataGearをALLOCATEするとData Tableのヒープ領域にそのDataGear分の領域が確保される．これらの複数のContextはContextキューで管理され，順次実行される．

RedBlackTreeは別のContextへコピーすると良い．同じContextへコピーする場合，GCはFromとToの領域が同じヒープ領域に置かれることになる．ヒープ領域を共有してしまうと，FromからToへの切り替えがしづらい．また，すでにそのヒープ領域でフラグメンテーションが起きている可能性があり，Copying GCの利点であるコンパクションができない．レプリケーションの場合，別のノードでは別のContextが動いているため，同じContextにコピーできるだけではレプリケーション機能は実現できない．別のContextへコピーすると，GCの場合，FromとToの領域が別のヒープ領域に置かれる．そのためContextを切り替えることによってFromからToへの切り替えができ，まっさらな状態のヒープ領域にコピーを行えるため，リニアアロケーションをしていればコンパクションが発生する．

動的にバックアップする際のコピーを考える．別 Context へコピーする場合，ディスクを Context の Data Table のヒープ領域と捉え，単純にディスク書き込みを行う．RedBlackTree は非破壊であり，複数のルートノードによってバージョン管理されている．そのため，あるバージョンのバックアップをリストアしたい場合がある．現在のバージョンをディスク上にある特定のバージョンに切り替える場合，現在のバージョンへの書き込みの最中にバージョンが切り替わってしまうとデータの一貫性を損なう．そのため，データリストア時の一貫性を確保する仕組みが必要となる．また，バージョン付きのバックアップは無尽蔵にデータが増加していく問題がある．それは，任意のタイミングでデータを区切ることで解決できる．仮想的に Data Table を作成し，そのヒープ領域 (ディスク) に対してデータをコピーする．過去の Data Table のバージョンに戻りたい場合があるため，Data Table は前の Data Table の参照を持つ．

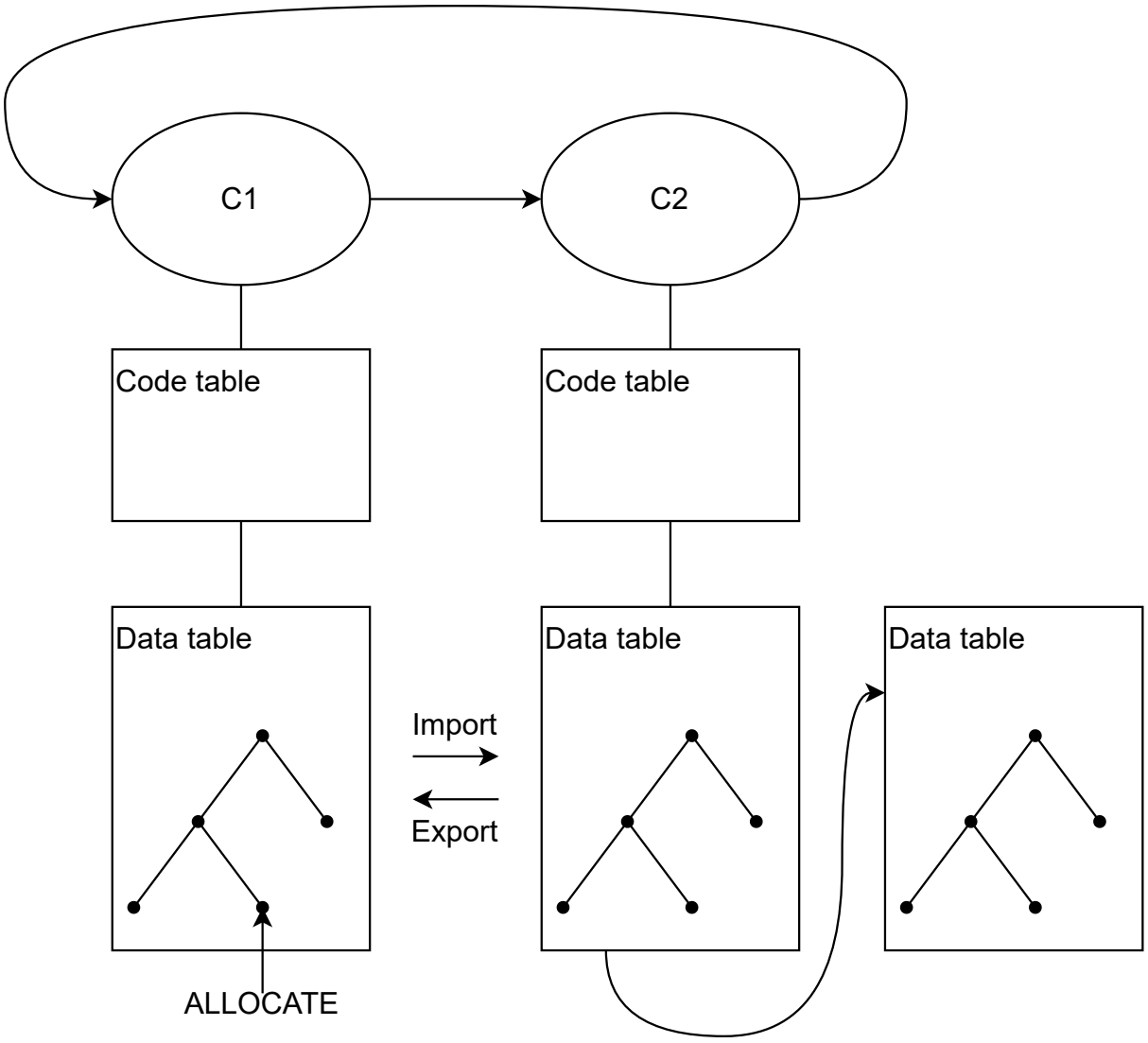

図 5.4: 別 Context へのコピー

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．

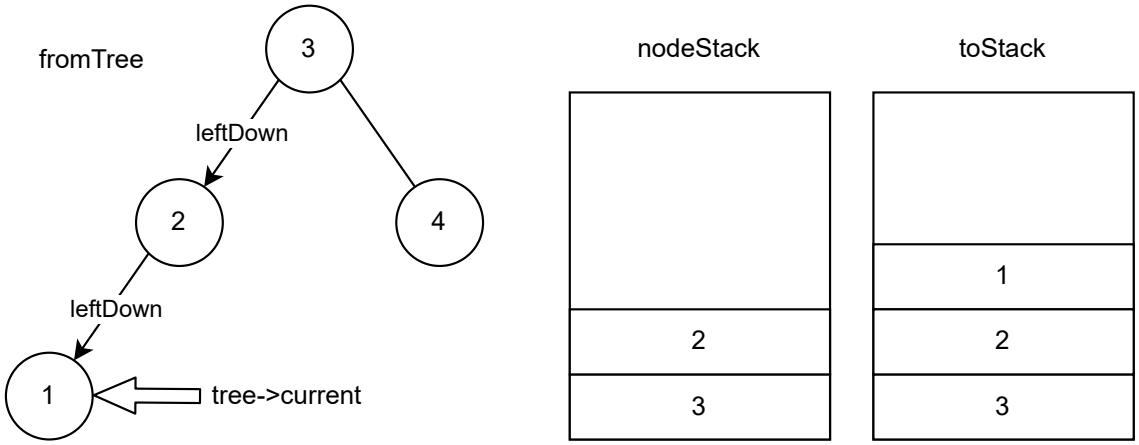

図 6.1: コピー元の Tree と 2 つの Stack の状態の例

6.3　アロケーション部分

新規ノードのアロケーションは copyRedBlackTree，leftDown，rightDown CodeGear で行われる．例としてソースコード 6.3 に leftDown1 CodeGear を示す．9 行目でコピー先の Node を new でアロケーションし，11〜14 行目でコピー元からコピー先のノードへ key，value，color をコピーしている．また，10 行目の data は toStack の top から得たもので，アロケーションしたノードの親ノードにあたる．17 行目は `data->left` にアロケーションしたノードを代入している．そのようにしてアロケーションしたノードは toStack に push され，次に leftDown で子ノードがアロケーションされる際に参照される．このようなアロケーションは copyRedBlackTree，rightDown においても同様に行われている．アロケーションはノーマルレベルでは new キーワードによって記述され，メタレベルではソースコード 3.9 で示した `ALLOCATE` マクロを呼び出すことによって行われる．

ソースコード 6.3: leftDown1 CodeGear(アロケーション部分の例)

```
1  __code leftDown1(struct RedBlackTree* tree, struct Stack* stack) {
2      printf("leftDown1\n");
3
4      if (tree->current->left == NULL) {
5          goto rightDown();
6      }
7
8      struct Stack* toStack = tree->toStack;
9      struct Node* newNode = new Node();
10     struct Node* data = (Node*)(stack->data);
11     newNode->key = tree->current->left->key;
12     newNode->value = (union Data*)new Integer();
13     ((Integer*)newNode->value)->value = ((Integer*)tree->current->left->
       value)->value;
```

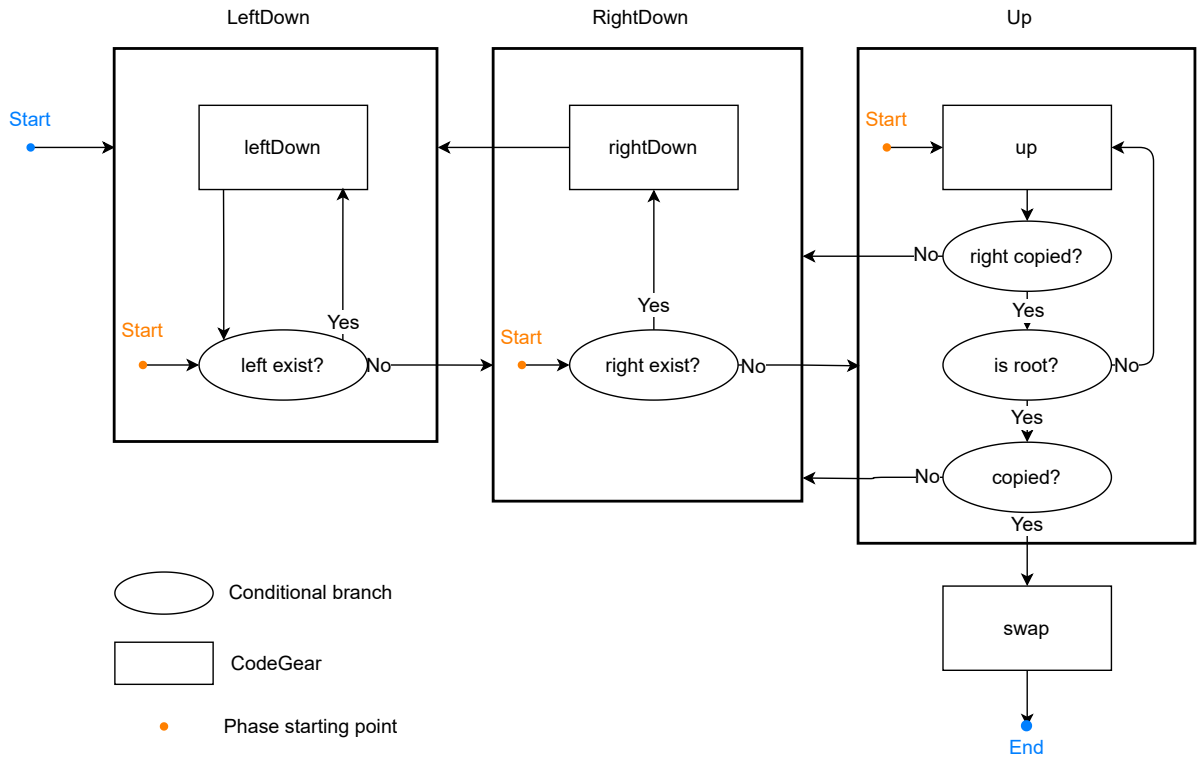

図 6.2: Copy 時の CodeGear の大まかな遷移

```
14      newNode->color = tree->current->left->color;
15
16      if(data) {
17          data->left = newNode;
18      }
19
20      goto toStack->push(newNode, leftDown2);
21  }
```

ソースコード 6.4 にビルド時に生成された，new に対応する ALLOCATE マクロの呼び出し部分を示す．&ALLOCATE に context と，アロケートしたい DataGear の型名を渡している．context は現在の Context を指しており，別の Context へのアロケーションはされていない．

ソースコード 6.4: ビルド時に生成された ALLOCATE 部分

```
1   struct Node* newNode = &ALLOCATE(context, Node)->Node;
```

6.4　swap

swap はコピー自体に関する機能ではなく，Copying GC の動作を想定した機能を実装したものである．Copying GC が From 領域と To 領域を入れ替えるように，コピー前後の木を入れ替える動作をする．swap は実際には swap, swap1, swap2 のような複数の CodeGear で構成されている．ソースコード 6.5 に木の入れ替え処理を行う swap2 CodeGear を示す．3 行目の toTree は，toStack に push した新しい木のルートノードに当たるノードである．4 行目で新規に RedBlackTree を作成し，5 行目で toTree を newRedBlackTree の root に設定している．その後，7 行目で tree に newRedBlackTree を指定することで木の入れ替えを行なっている．これらの動作は同一の Context 上で行われている．図 6.3 で示すように，Context の Data Table 上の Tree DataGear がもつ RedBlackTree を同ヒープの From RedBlackTree から To RedBlackTree へ参照を入れ替える．

ソースコード 6.5: swap2 CodeGear(木の入れ替え処理部分)

```
1  __code swap2(struct RedBlackTree* tree, struct Stack* stack, __code next
       (...)) {
2      printf("swap2\n");
3      struct Node* toTree = (Node*)(stack->data);
4      struct RedBlackTree* newRedBlackTree = new RedBlackTree();
5      newRedBlackTree->root = toTree;
6      newRedBlackTree->current = toTree;
7      tree = newRedBlackTree;
8
9      printf("copied\n");
10
11     goto next(...);
12 }
```

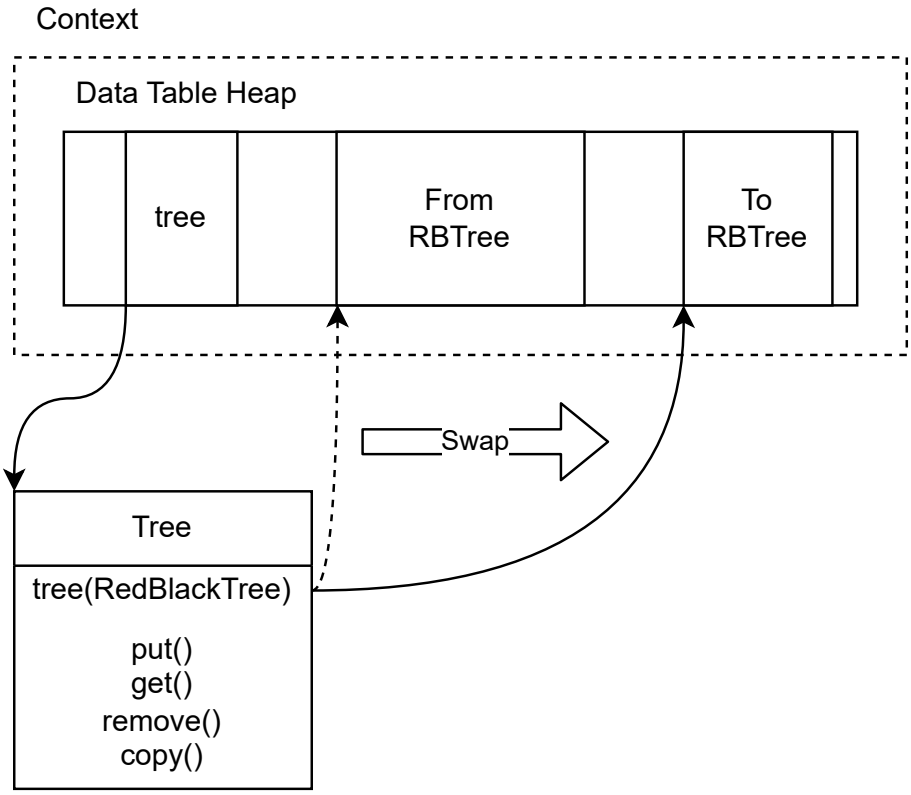

図 6.3: swap 時の Tree DG と Data Table の様子

第7章　実装の評価

7.1　非破壊RedBlackTreeの増大抑制

今回はCopyRedBlackTreeの実装によって，コピー先の木はコピー時にrootから参照可能であったノードのみを持ち，参照されていないノードが排除されているため，コピーをすることで参照する木の増大を防ぐことが可能である．しかし，コピー元の木やゴミのメモリ領域を再利用する仕組みがないため，メモリ領域の使用量の軽減にはつながっていない．現状は同一Context上のCode Tableのヒープ領域に木をコピーしている．別Contextへ木をコピーし，コピー元のContext自体を廃棄する仕組みを実装することでメモリ領域の再利用が可能であると考える．

7.2　テストコード

CopyRedBlackTreeのプログラムを作成する際に，図??のような11パターンの木の状態と10000ノードの木をコピーするテストコードを作成した．11パターンの木の状態においてコピーは動作した．10000ノードの木のコピーはヒープオーバーフローによるsegmentation fault errorが発生する．3616ノードでは動作するため，ヒープ領域の容量が不足していると考えられる．この事象を解決するためには，ALLOCATEする際にヒープ領域の容量不足を検知し，ヒープ領域の拡張を行うことが考えられる．これらのテストコードによってある程度の動作確認は可能であるが，完全に正しい動作をしているかどうかの証明はできていない．

7.3　RedBlackTreeの持続性

CopyRedBlackTreeは木を単純にコピーする機能であるため，RedBlackTree乃至，データの持続性が確保されている．

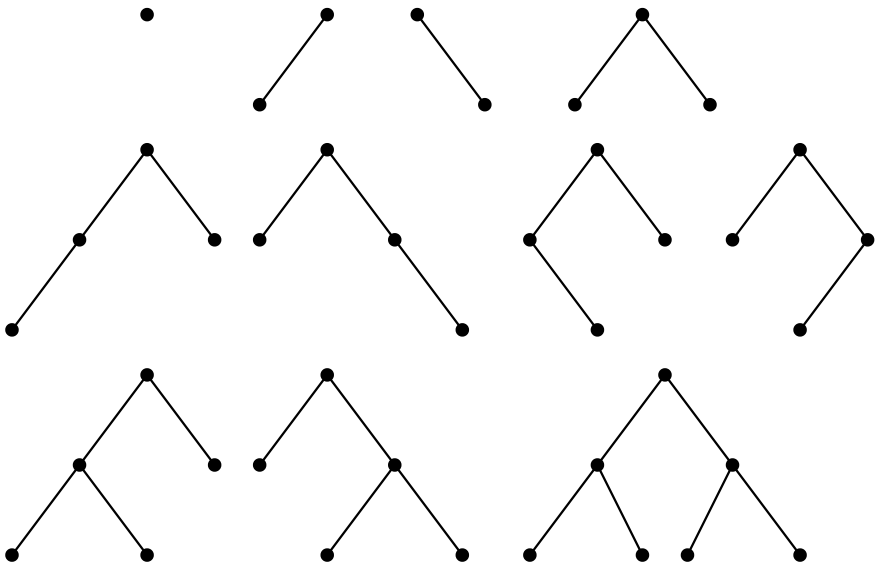

図 7.1: CopyRedBlackTree のテストパターン

7.4　Stack の使用

今回の実装では SingleLinkedStack を 2 つ使用している．CbC は call stack などの状態を持たないことによる利点があるため，Stack を使用することによってその利点を得られなくなってしまうことが考えられる．しかしながら，今回の Stack は Call stack とは違い，プログラムの中で明示的に使用され，また，CodeGear 自体が状態を持たないため問題がない．

第8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．

8.1　別 Context への ALLOCATION

今回構築した copyRedBlackTree では，木を同じ Context 上のヒープ領域にコピーする．しかし，別ノードへのコピーを実現するには別 Context へコピーを行う必要がある．現状は，ビルド時にメインの Context と ALLOCATION のマクロを生成するため，別の Context を指定することができない．そのため，ALLOCATION に別 Context を操作するための機能を導入する必要があると考える．

8.2　ヒープオーバーフロー問題

RedBlackTree は Context 上の Data Table のヒープ領域に確保，コピーされる．この際，ヒープ領域の容量が足りず，ヒープオーバーフローする問題がある．ヒープ領域は Context 生成時に確保され，ALLOCATE マクロによって DataGear が配置される．ALLOCATE は DataGear をヒープ領域に配置するのみであるため，容量が足りない場合にヒープ領域を拡張する機能を追加する必要がある．

8.3　テストケースの生成

CopyRedBlackTree を作成する際，コピーの動作確認のためのテストケースを 11 パターン用意した．その際のテストコードの行数が約 1000 行となった．開発の効率化のため，より簡潔にテストコードを書けるような仕組みを作る必要があると考える．

8.4　GC とレプリケーションの実装

今回は GC やレプリケーションの実装をするために必要な木のコピー機能である CopyRedBlackTree の実装を行った．Copying GC の動作を想定した swap を作成したものの，コンパクションが行われないなど GC として不十分な点がある．今後は十分な機能をもった GC やレプリケーションを実装していきたい．GC は Context の廃棄やフリーリスト，GC タイミングの機構が必要である．また，GearsOS 全体を GC することも考えられる．レプリケーションは別ノードでのコピーの実行やレプリカとの通信プロトコルを定義する必要があるだろう．

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.

[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] Ryan M. Golbeck, Samuel Davis, Immad Naseer, Igor Ostrovsky, and Gregor Kiczales. Lightweight virtual machine support for aspectj. In *Proceedings of the 7th International Conference on Aspect-Oriented Software Development*, AOSD '08, p. 180–190, New York, NY, USA, 2008. Association for Computing Machinery.

[10] 清水隆博. Gearsos のメタ計算. 修士 (工学) 学位論文, March 2021.

[11] 並列信頼研究室. Gearsos. http://www.cr.ie.u-ryukyu.ac.jp/hg/Gears/Gears/.

[12] 伊波立樹. Gearsos の並列処理. 修士 (工学) 学位論文, March 2018.

[13] 並 列 信 頼 研 究 室. Gearsagda. http://www.cr.ie.u-ryukyu.ac.jp/hg/Gears/GearsAgda/.

[14] 河野 真治 (琉球大学) 森 逸汰. Gearsagda による red black tree の検証, May 2023.

[15] 並列信頼研究室. Cbc_xv6. http://www.cr.ie.u-ryukyu.ac.jp/hg/CbC/CbC_xv6/.

[16] 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.

[17] 河野 真治 (琉球大学) 仲吉 菜々子. Gears os の codegear management, May 2023.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] Rajeev Joshi and Gerard J. Holzmann. A mini challenge: build a verifiable filesystem. *Form. Asp. Comput.*, Vol. 19, No. 2, p. 269–272, jun 2007.

[23] Jing Liu, Anthony Rebello, Yifan Dai, Chenhao Ye, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Scale and performance in a filesystem semi-microkernel. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, p. 819–835, New York, NY, USA, 2021. Association for Computing Machinery.

[24] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Trans. Storage*, Vol. 9, No. 3, aug 2013.

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