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[,a)]　河野 真治[,b)]

概要：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．現在，GearsOSには未実装の機能がいくつかあり，その一つとしてファイルシステムが挙げられる．それは，OS上でアプリケーションを動作させるために，特に重要な機能であり，必要不可欠であるため実装したい．今回，GearsOSのファイルシステムを実装するにあたり，Unixのinodeの仕組みを参考にした．また，ファイルシステムと密接な関係であるメモリマネージメントについて考察した．

Building a Filesystem using inode in GearsOS

1. GearsOSにおけるファイルシステム

アプリケーションの信頼性を保証することは情報システムやコンピュータを用いる業務の信頼性の保障につながる重要な課題である．したがって，アプリケーションの信頼性を保証するために，基盤となるOSの信頼性を高める必要がある．

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，OSの信頼性を定理証明やモデル検査を行うことで保証することを目指している[1]．同じく，当研究室で開発しているプログラム言語であるCbC（Continuation based C）で記述されており，ノーマルレベルとメタレベルを簡単に切り分けることを可能としている．そのようにして，CbCでメタレベルの処理を切り出したものに対して，定理証明やモデル検査を行うことで信頼性を保証する．

GearsOSは現在OSとして重要な機能がいくつか未実装であり，その一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ファイルやディレクトリといった構造を持ち，データの保存，整理を行う．また，OSが管理するデータの操作を人間が行いやすいようにインターフェースを提供する．OSの機能の中でも特に重要な機能であるため，GearsOSにも実装を行う必要がある．

今回GearsOSへファイルシステムを実装するにあたり，Unixのファイルシステムを参考にした．Unixのファイルシステムではファイルのメタデータをinodeの形式で保持している．同様に，inodeの仕組みを用いてGearsOSのファイルシステムを実装したい．また，インターフェースについても，cd，ls，mkdirというようにUnix likeに実装したい．当研究室ではxv6のCbCでの書き換えを行なっているが，今回はxv6のルーチンをCbCで書き換えるのではなくGearsOSへUnixのファイルシステムの仕組みを取り入れるアプローチをとりたい．それはGearsOSとCbCで書き換えたxv6の比較や，互いにファイルシステムの機能の移植が行える様にするためである．

GearsOSのファイルシステムを構築するにあたり，メモリマネージメントについて考察する．現在，GearsOSにはメモリマネージメントのシステムが存在しない．しかしながら，ファイルシステムを構築するにあたりメモリマネージメントが必要不可欠である．メモリ上のRedBlackTreeで構築されたデータ構造をそのままディスクにコピーする形で実装することを目指したい．

2. Continuation based C

Continuation based C（CbC）[2], [3]は，当研究室で開発しているCの下位言語である．CbCでは関数の代わりにCodeGearという単位でプログラミングを行う．CodeGearは*__code*という記述で宣言することができる．また，データ

[1] 情報処理学会
IPSJ, Chiyoda, Tokyo 101–0062, Japan
[†1] 現在，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
Presently with University of the Ryukyus, Graduate School of Engineering and Science
[a)] matac@cr.ie.u-ryukyu.ac.jp
[b)] kono@ie.u-ryukyu.ac.jp

の単位には DataGear と呼ばれる変数データを用いる．図 1 は CodeGear と入出力の関係を表している．CodeGear は DataGear を入力として受け取り，別の DataGear に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 に遷移することができ，その際，Output DataGear を次の CodeGear の Input DataGear として渡すことができる．

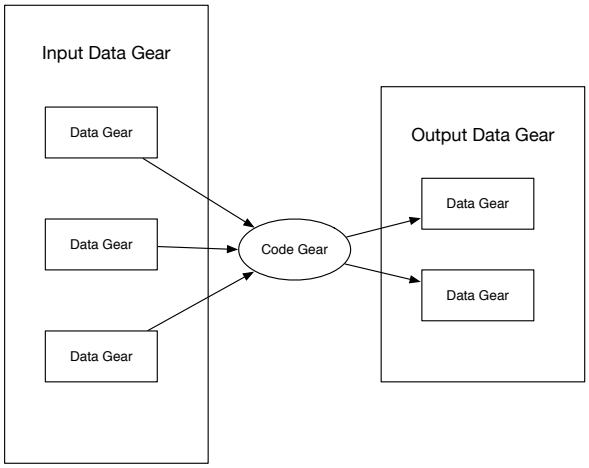

図 1: CodeGear と入出力の関係図

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 のような環境を保存しない．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．これらの仕組みにより，ノーマルレベルとメタレベルの処理を容易に切り分けることが可能となる．

CbC のプログラム例をソースコード 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．

Code 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);
}

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

3. GearsOS

GearsOS[4], [5], [6] 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．

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 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 は CodeGear の遷移と MetaCodeGear の関係を表

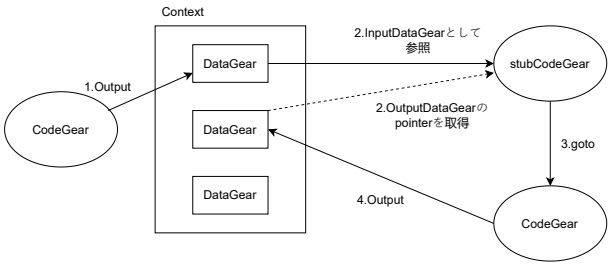

図 2: Context を参照する流れ

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．

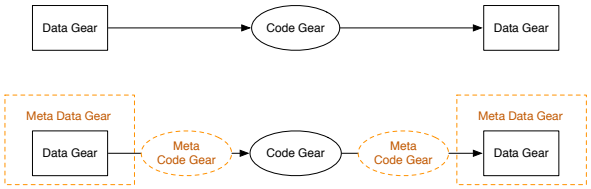

図 3: CodeGear と MetaCodeGear の関係

4. Unix のファイルシステム

Unix のファイルシステムは BTree と inode で構成されており，xv6 もその仕組みを用いている．xv6[7] は MIT で教育用の目的で開発された OS で，Unix の基本的な構造を持つ．当研究室では xv6 の CbC での書き換え，分析を行なっている [8], [9].

inode 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 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．

当研究室では xv6 の CbC での実装を行なっているが，

File Types	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 1: inode でのファイル属性情報

今回は xv6 の File ルーチンを CbC で書き換えるのではなく GearsOS へ Unix のファイルシステムの仕組みを取り入れるアプローチをとる．まず，ファイルシステムを大まかにディレクトリシステムとファイルの二つに分けて考える．ディレクトリシステムは Unix の inode の仕組みを取り入れる．今回作成した，GearsOS のディレクトリシステムである GearsDirectory について説明する．ファイルについては後の章で述べる．

FileSystemTree はディレクトリ構造，inode の仕組みを取り入れる際に用いる Tree である．ソースコード 2 は FileSystemTree の interface である．GearsOS における interface は CodeGear と各 CodeGear が用いる I/O DataGear の集合を記述する．よって，FileSystemTree の interface は fTree と node の DataGear と put, get, remove, next の CodeGear を持つ．FileSystemTree の fTree は GearsOS の永続データを構築する際に使用される RedBlackTree であり，put, get, remove は RedBlackTree の操作を行うための CodeGear である．また，next は遷移先の CodeGear を参照するために用いる．

Code 2: FTree の interface

```
typedef struct FTree<>{
    union Data* fTree;
    struct Node* node;
    __code put(Impl* fTree,Type* node, __code next
        (...));
    __code get(Impl* fTree, Type* node, __code next
        (...));
    __code remove(Impl* fTree,Type* node, __code next
        (...));
    __code next(...);
} FTree;
```

ディレクトリ構造は 2 つの FileSystemTree で実装する．1 つ目は inode number と file のポインタのペアを持つ木である．それは，inode number を key，inode を value として持つため inode number から inode を検索するために用いる（以下，inode tree とする）．2 つ目は filename と inode number のペアを持つ木である．それは，filename を key, inode number を value として持つため，filename から inode number を検索するために用いる．また，inode を

filename で検索するための index tree であるといえる（以下，index tree とする）．

図 4 は index tree を用いた inode の検索の流れを表す．まず index tree から key が filename の node を get する．key が filename の node の value より inode number がわかる．次に，取得した inode number を key として inode tree を検索する．key が inode number の node は value として inode を持っていて，inode を参照することができる．

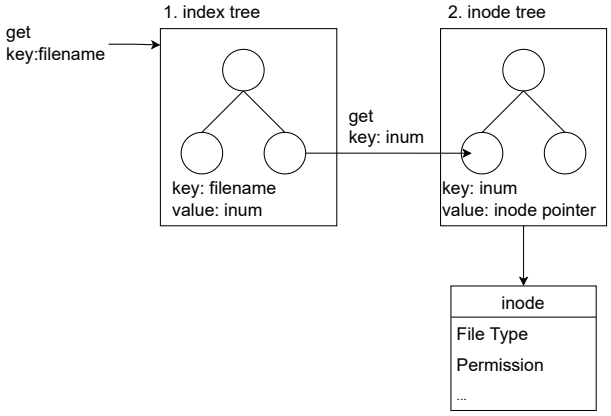

図 4: index tree を用いた inode の検索の流れ

GearsOS における永続データは非破壊的な編集を行う木構造を用いて保存する．図 5 は非破壊的編集を木構造に対し行う様子である．赤で示されたノード 6 を A に編集する場合，まずルートノードから編集ノードまでのパスを全てコピーする．コピーしたパス上に存在しないノードは，コピー元の木構造と共有する．それにより，編集後の木構造の赤のルートノードから探索を行う場合は編集された A のノードが見え，黒のルートノードから探索を行う場合は編集前の 6 のノードを見ることができる．ディレクトリシステムを非破壊的な木構造の編集を用いて実装することにより，ディレクトリシステム自体にバックアップの機能を搭載することが可能であると考える．

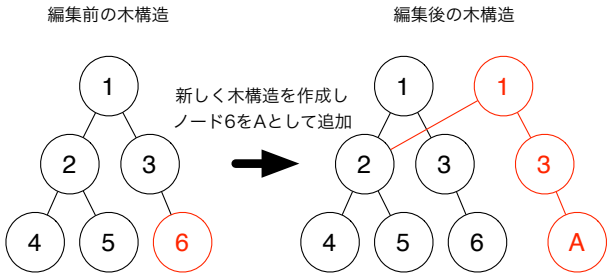

図 5: 非破壊的な Tree 編集

5. GearsFileSystem におけるインターフェース

ファイルやディレクトリの操作を行うインターフェースを Unix Like に実装を行った．実装を行った mkdir, ls, cd を説明する．

5.1 mkdir

Unix において mkdir は新しくディレクトリを作成するコマンドである．GearsDirectory の mkdir は index tree と inode tree に node を put することでディレクトリを作成する．ソースコード 3 は GearsDirectory における mkdir の CodeGear であり，図 6 はその処理を図で表したものである．まず 1 行目の*__code mkdir* では inode tree へ inode の put が行われ，*__code mkdir2* へ遷移する．inode は 4,5 行目で key に inode number, value にディレクトリのポインタがセットされる．次に，11 行目の*__code mkdir2* では index tree へ key が filename，value が inode number の node の put が行われ，next の CodeGear へ遷移する．このように，FileSystemTree の put を 2 回行うため，mkdir は*__code mkdir* と*__code mkdir2* の 2 つの CodeGear で構成されている．また，InputDataGear の *name* は filename を表す．

Code 3: mkdir の CodeGear

```
__code mkdir(struct GearsDirectoryImpl*
     gearsDirectory, struct Integer* name, __code next
     (...)) {
    struct FTree* newDirectory = createFileSystemTree(
         context, gearsDirectory->currentDirectory);
    Node* inode = new Node();
    inode->key = gearsDirectory->INodeNumber;
    inode->value = newDirectory;
    struct FTree* cDirectory = new FTree();
    cDirectory = gearsDirectory->iNodeTree;
    goto cDirectory->put(inode, mkdir2);
}

__code mkdir2(struct GearsDirectoryImpl*
     gearsDirectory, struct Integer* name, __code next
     (...)) {
    Node* dir = new Node();
    dir->key = name->value;
    Integer* iNum = new Integer();
    iNum->value = gearsDirectory->INodeNumber;
    dir->value = iNum;
    gearsDirectory->INodeNumber = gearsDirectory->
         INodeNumber + 1;
    struct FTree* cDirectory = new FTree();
    cDirectory = gearsDirectory->currentDirectory;
    goto cDirectory->put(dir, next(...));
}
```

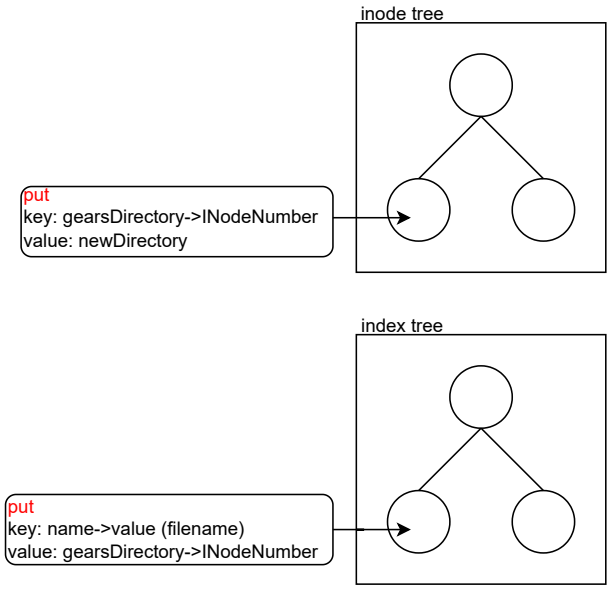

図 6: mkdir の操作の流れ

5.2 ls

Unix において ls はファイルやディレクトリの一覧，メタ情報を表示するコマンドである．GearsDirectory の ls は index tree に対し，get をすることでディレクトリの name を取得する．これは，Unix の ls コマンドにおける *$ls filename* に等しい機能である．ソースコード 4 は GearsDirectory における ls の CodeGear であり，図 7 はその処理を図で表したものである．まず 1 行目の*__code ls* では index tree に対し get を行うため，3 行目で get したい filename を key にセットし，index tree の get へ goto している．その後，9 行目の*__code ls2* では node->key に格納された get の結果を printf で出力する．本来 ls コマンドは引数を渡さずに実行するとカレントディレクトリ下のディレクトリやファイルを一覧で表示するが，現時点では未実装である．なお，一覧表示の機能は filename のリストをディレクトリに持たせることで実装可能であると思われる．

Code 4: ls の CodeGear

```
__code ls(struct GearsDirectoryImpl* gearsDirectory,
     struct Integer* name, __code next(...)) {
    Node* dir = new Node();
    dir->key = name->value;
    struct FTree* cDirectory = new FTree();
    cDirectory = gearsDirectory->currentDirectory;
    goto cDirectory->get(dir, ls2);
}

__code ls2(struct GearsDirectoryImpl* gearsDirectory,
      struct Node* node, __code next(...)) {
    printf("%d\n", node->key);
    goto next(...);
}
```

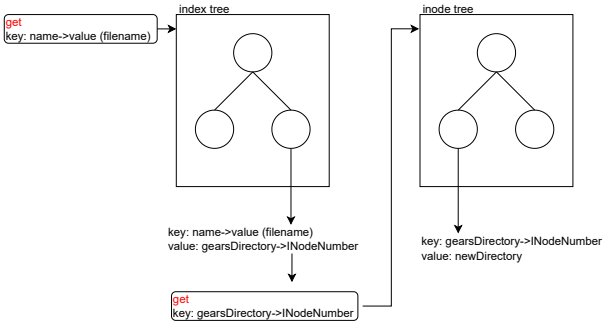

図 7: ls の操作の流れ

5.3 cd

Unix において cd はディレクトリを移動するコマンドである．GearsDirectory の cd は index tree と inode tree に対し get を行い，currentDirectory を書き換えることで実装する．機能としてはディレクトリが持つ子ディレクトリへの移動ができる．ソースコード 5 は GearsDirectory における cd の CodeGear であり，図 8 はその処理を図で表したものである．まず 1 行目の*__code cd2Child* で index tree に対し get を行うため，index tree の get へ goto している．次に，9 行目の*__code cd2Child2* で inode tree に対し get を行うため，inode tree の get へ goto している．この際，get は 1 行目の cd2Child で get した node の value をもとに行う．value には inode number がセットされている．その後，15 行目の*__code cd2Child3* で current ディレクトリを保存している gearsDirectory->currentDirectory を get した node->value に書き換える．

Code 5: cd の CodeGear

```
__code cd2Child(struct GearsDirectoryImpl*
     gearsDirectory, struct Integer* name, __code next
     (...)) {
    struct FTree* cDirectory = new FTree();
    cDirectory = gearsDirectory->currentDirectory;
    struct Node* node = new Node();
    node->key = name->value;
    goto cDirectory->get(node, cd2Child2);
}

__code cd2Child2(struct GearsDirectoryImpl*
     gearsDirectory, struct Node* node, __code next
     (...)) {
    struct FTree* iNodeTree = new FTree();
    iNodeTree = gearsDirectory->iNodeTree;
    goto iNodeTree->get(node->value, cd2Child3);
}

__code cd2Child3(struct GearsDirectoryImpl*
     gearsDirectory, struct Node* node, __code next
     (...)) {
    gearsDirectory->currentDirectory = node->value;
    goto next(...);
}
```

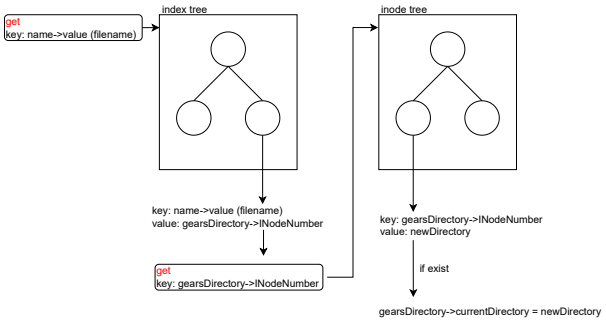

図 8: cd の操作の流れ

6. GearsFileSystem におけるファイルの構成

ファイルシステムはディレクトリの構成だけでなく，ファイルの構成についても考える必要がある．本研究と並行する形で一木貴裕による分散ファイルシステムの設計が行われており [10]，ファイルの構成に関しても実装，検討されている．GearsOS におけるファイル構成を説明する．

ファイルの Input/Output Stream は競合的なアクセスに対応するため，3 つの SynchronizedQueue を用いる．それぞれを InputQueue, OutputQueue, mainQueue と呼ぶ．データを input したい場合 InputQueue へ put を行い，取得したい場合 OutputQueue から get を行う．mainQueue はデータそのものであり，InputQueue から mainQueue, mainQueue から OutputQueue へデータが流れるように接続される．これらは，Gear の概念では DataGear にあたり，DataGearManager に key と I/O Queue が対応する形で保持される．ファイルの中身のデータをレコードに分割し，レコードを Queue に put して stream に入れる．データを取り出す際は Queue から get し，順番に読むことでファイルを構築する．

分散ファイルシステムはファイルのデータ送受信をする際に用いられる API を作成する必要がある．WordCount 例題 [11] を通して，GearsFile の API の作成を行う．WordCount 例題は指定したファイルの文字数や行数，ファイルの内の文字列を出力する．図 9 は WordCount 例題の処理の流れを示している．これは大きく分けて，指定したファイルを File 構造体として open する FileOpen スレッド，File 構造体を受け取り文字数と行数を countUp する WordCount スレッドの二つの CodeGear で記述することができる．また，ファイル内の文字列を行ごとに CountUp に送信し，EOF を受け取ったらループを抜け finish に移行する．

7. GearsOS のメモリマネージメントシステムの考察

GearsOS のメモリマネージメントは，メモリ上のデータとディスク上のデータの構造が等しくなる形で実装をしたい．メモリ上とディスク上でデータの構造を等しくすることで，単純なコピーでメモリとディスク間のデータのやり取りを行うことができる．よって，比較的簡素に実装を行うことができると予想する．しかしながら，メモリ上とディスク上でオフセットの差が出る問題がある．これは，メタ計算でオフセットの差を吸収する処理を行うことで解決させる．また，メモリ上とディスク上のデータのアドレスが異なるため，ユーザーレベルからポインタを用いた場合，問題になる．しかしながら，GearsOS ではユーザーレベルでポインタを用いることを禁止しているため，問題ないと考える．

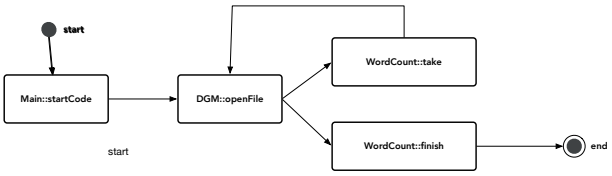

図 9: WordCount with CbC

ガベージコレクションについては，Copying GC を用いる．Copying GC は単純に用いるとメモリ容量が倍必要になるという問題がある．そこで，リンクしているデータのみをコピーすることによってメモリ使用量を削減したい．データがリンクされているかどうかは LinkedList を参照し判断する．

8. 今後の課題

8.1 GearsShell

GearsOS に存在する未実装の機能の一つに shell が挙げられる．現状の GearsOS はユーザーの入力を受け付けることが出来ず，プログラミングインターフェースの様に機能している．GearsFileSystem など GearsOS の各機能と接続し，今回作成した cd や ls の様なコマンドを受け付ける GearsShell を作成したい．

8.2 GearsDirectory filename

現状は GearsDirectory の filename は Integer の構造で管理されている．しかしながら，filename は一般的に文字列型であるため Integer から文字列型に変更する必要がある．

8.3 GearsDirectory path

GearsDirectory には path の機能が実装されていない．よって full path 指定の ls などが実装できない状態である．FileSystemTree を拡張し，ノードをたどり path を生成する様な機能を実装する必要がある．

8.4 ファイルのバックアップ

ディレクトリに関しては非破壊的な Tree 編集を用いることで，バックアップを行うことを考えたがファイルに関

してはレコードの Data をファイルの差分履歴として保持し，日時情報を付け加えることで Version Control System のような機能を持たせることが可能であると考えられる．

8.5 GearsDirectory on disk

現状は GearsDirectory の inode は on memory で実装されている．データの保存は disk 上に行うため，inode を disk 上に構築し必要がある

9. まとめ

本研究では主として GearsFileSystem の構築に必要な GearsDirectory の実装について説明した．いくつか課題はあるが，RedBlackTree のシンプルなインターフェースにより比較的容易に実装を行うことができた．また，RedBlackTree を用いて inode の仕組みを構築し，ls，cd，mkdir を作成するなどして，Unix Like に構築することが出来た．メモリマネージメントについては，今後の研究で実装と考察を行い，Gears OS のメモリマネージメントシステムを構築していく．

信頼性については，定理証明やモデル検査を用いて保証を行うが，非破壊的な Tree 編集によるディレクトリのバックアップやファイルのバックアップをファイルシステムに組み込むことでも信頼性の向上が期待できる．形式手法とファイルシステムの機能の両面で信頼性の向上が図れると考える．

参考文献

[1] 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) (2020).
[2] 並列信頼研究室：CbC, http://www.cr.ie.u-ryukyu.ac.jp/hg/CbC/CbC_llvm/.
[3] 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 (2000).
[4] 清水隆博：GearsOS のメタ計算，修士 (工学) 学位論文 (2021).
[5] 並列信頼研究室：GearsOS, http://www.cr.ie.u-ryukyu.ac.jp/hg/Gears/Gears/.
[6] 伊波立樹：GearsOS の並列処理，修士 (工学) 学位論文 (2018).
[7] 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.
[8] 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) (2020).
[9] 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) (2019).
[10] 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 (2022).
[11] 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) (2021).
[12] 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 (2018).