2022年度 卒業論文

Bachelor's Thesis

GearsOSにおけるinodeを用いたFileSystemの設計

Designing a FileSystem in GearsOS

琉球大学工学部工学科知能情報コース

185742J　又吉　雄斗

指導教員　河野　真治

要旨

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．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．信頼性を保証するための手法として定理証明やモデル検査が挙げられる．

当研究室では，定理証明やモデル検査による信頼性の保証を目的とした GearsOS を開発している．GearsOS はノーマルレベル，メタレベルの処理を切り分けることができる Continuation Based C(CbC) で記述されており，Gear というプログラミング概念を持つ．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 には現在未実装の機能があり，その一つにファイルシステムが挙げられる．信頼性をモデル検査を通して保証することができる GearsOS のファイルシステムを実装したい．

本論文では，まず GearsOS のファイルシステム構築に関する基礎概念の紹介をし，その後 GearsOS のファイルシステムの具体的な構築方法について述べる．

Abstract

Ensuring the reliability of applications is an important issue for ensuring the reliability of information systems and operations that use computers. In order to guarantee the reliability of applications, it is necessary to improve the reliability of the underlying operating system. Theorem proving and model checking are the methods to guarantee the reliability.

In our laboratory, we are developing GearsOS for the purpose of guaranteeing reliability by theorem proving and model checking. GearsOS is written in Continuation Based C (CbC), which can separate normal-level and meta-level processing, and has the programming concept of Gear. GearsOS is written in Continuation Based C (CbC), which can separate normal-level and meta-level processing.

There are currently unimplemented features in GearsOS, one of which is the file system. We would like to implement a file system for GearsOS that can guarantee reliability through model checking.

In this paper, we first introduce the basic concept of the file system construction of GearsOS, and then describe the concrete construction method of the file system of GearsOS.

目次

図目次

表目次

ソースコード目次

第 1 章

Continuation based C

Continuation based C(CbC)[**?**, **?**] とは C の下位言語である．function call の代わりに goto による継続を用いる．CbC のプログラムは CodeGear と呼ばれる処理の単位で記述し，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可能である．

1.1 CodeGear と DataGear

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．データの単位には DataGear と呼ばれる変数データを用いる．図 1.1 は CodeGear と DataGear 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 1.1　非破壊的な Tree 編集

1.2 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．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 の継続は関数の繊維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 における継続と区別して，軽量継続と呼ぶ．

1.3 CbC の記述例

CbC のプログラム例をソースコード 1.1 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 1.1 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 2 章

GearsOS

GearsOS[**?**, **?**, **?**] は 信頼性と拡張性の両立を目的として，開発されている．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．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でき，同様に Gear の概念を持つ CbC(Continuation based C) で記述されている．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．

2.1 stubCodeGear

図 2.1 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．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．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．

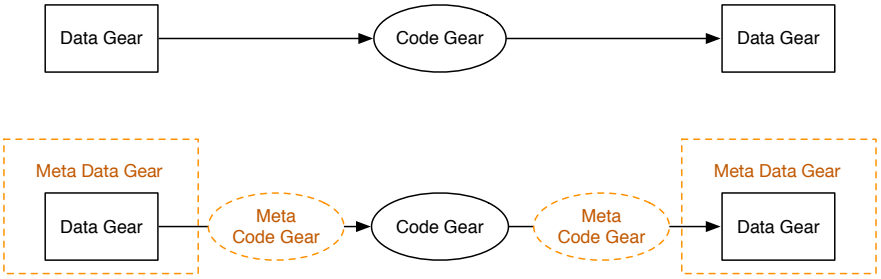

図 2.1 CodeGear と MetaCodeGear の関係

2.2 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．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．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 2.2 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 し，次の stubCodeGear へ Context を参照し goto を行う．この際直接参照はせず，goto meta を経由し goto を行う．stubCodeGear は InputDataGear(前の CodeGear の OutputDataGear) と OutputDataGear を参照し，次の CodeGear へ goto を行う．CodeGear での処理後，OutputDataGear へデータを output し，次の stubCodeGear へ goto を行う．

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．

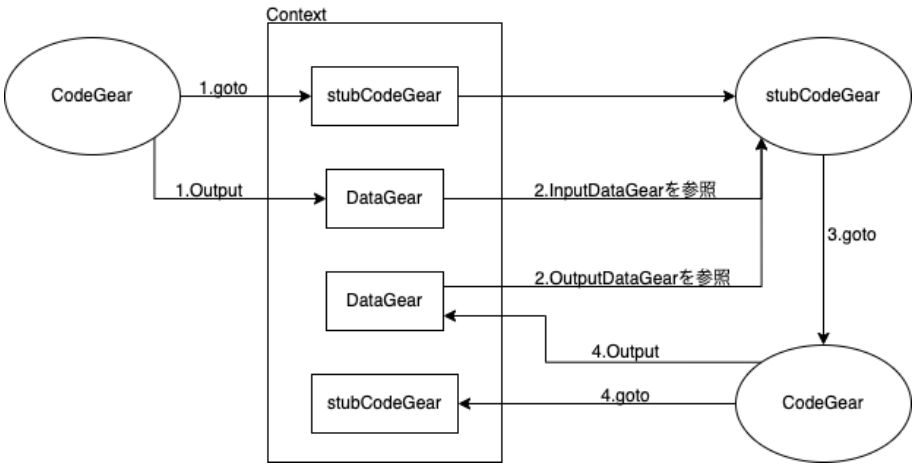

図 2.2 Context を参照する流れ

第 3 章

Christie

Christie は当研究室で開発を行っている Java で記述された分散フレームワークである．CbC と似ているが別物の Gear という概念や，任意の Topology を形成するための TopologyManager がある．

3.1 Gear の概念

Christie には以下の 4 つの Gear という概念が存在する．

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

CodeGear はクラスやスレッドに相当する．DataGear は変数データに相当し，Java のアノテーションを用いて記述される．CGM はいわゆるノードに相当し，CodeGear，DataGear，DGM を管理する．複数の CGM 同士が配線され，DataGear を送信し合うことで分散処理を実現している．DGM は DataGear を管理しているもので変数プールに相当する．

3.2 DataGearManager

DataGearManager は key value store の構造を持つ．CGM が利用する CG の key と put された DataGear の組み合わせで DataGear を管理する．DGM は LocalDGM と

RemoteDGM に区別することができる．LocalDGM は CGM 自身が所持する DataGear のプールである．ReomoteDGM は CGM が配線されている別の CGM がもつ DG のプールである．

3.3 Topology-Manager

Christie には任意の topology を生成し，ノード同士の通信接続を管理ことができる Topology-Manager という機能が存在する．Topology-Manager で生成できる topology には静的 topology と動的 topology の 2 つがある．静的 topology はプログラマが任意の topology とノードの配線を行うことができる．dot ファイルにノードの配線を記述し，Topology-Manager に参照させることで生成する．dot ファイルに記述したノードの数と参加ノードの数が一致した場合に動作する．動的 topology は参加を表明したノードに対し，自動的に配線を行う．

第 4 章

Unix の File system

4.1　xv6

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．xv6 は Unix 系の OS であるため，File system では *inode* の仕組みが用いられている．

4.2　inode

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

第 5 章

GearsFileSystem の directory

当研究室では xv6 の CbC での実装を行なっているが，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 File system の仕組みを取り入れるアプローチをとる．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．

5.1 FileSystemTree

FileSystemTree は今回 directory 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 5.1 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．FileSystemTree の interface は fTree と node の DataGear と put, get, remove, next の CodeGear を持つ．FileSystemTree の実体は RedBlackTree であり，put，get，remove は RedBlackTree の操作を行うための CodeGear である．next は遷移先の CodeGear

ソースコード 5.1　FTree の interface

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

5.2 Tree による directory 構造

ディレクトリ構造は 2 つの RedBlackTree 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．inode number を key，file pointer を value として持つため inode number から file pointer を検索するために用いる．2 つ目は filename と inode number のペアを持つ木である．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．

5.3 Unix Like な interface

ファイルやディレクトリの操作を行う interface を Unix Like に実装を行った．実装を行った mkdir, ls, cd を説明する．

5.3.1 mkdir

ソースコード 5.2 は mkdir の CodeGear である．

ソースコード 5.2　mkdir の CodeGear

```
1  __code mkdir(struct GearsDirectoryImpl* gearsDirectory, struct Integer
       * name, __code next(...)) {
2      struct FTree* newDirectory = createFileSystemTree(context,
       gearsDirectory->currentDirectory);
3      Node* inode = new Node();
4      inode->key = gearsDirectory->INodeNumber;
5      inode->value = newDirectory;
6      struct FTree* cDirectory = new FTree();
7      cDirectory = gearsDirectory->iNodeTree;
8      goto cDirectory->put(inode, mkdir2);
9  }
10
11 __code mkdir2(struct GearsDirectoryImpl* gearsDirectory, struct
       Integer* name, __code next(...)) {
12     Node* dir = new Node();
13     dir->key = name->value;
14     Integer* iNum = new Integer();
15     iNum->value = gearsDirectory->INodeNumber;
16     dir->value = iNum;
17     gearsDirectory->INodeNumber = gearsDirectory->INodeNumber + 1;
18     struct FTree* cDirectory = new FTree();
```

```
19      cDirectory = gearsDirectory->currentDirectory;
20      goto cDirectory->put(dir, next(...));
21  }
```

5.3.2 ls

ソースコード 5.3　ls の CodeGear

```
1   __code ls(struct GearsDirectoryImpl* gearsDirectory, struct Integer*
        name, __code next(...)) {
2       Node* dir = new Node();
3       dir->key = name->value;
4       struct FTree* cDirectory = new FTree();
5       cDirectory = gearsDirectory->currentDirectory;
6       goto cDirectory->get(dir, ls2);
7   }
8
9   __code ls2(struct GearsDirectoryImpl* gearsDirectory, struct Node*
        node, __code next(...)) {
10      printf("%d\n", node->key);
11      goto next(...);
12  }
```

5.3.3 cd

ソースコード 5.4　cd の CodeGear

```
1   // 仮
2   __code cd2Child(struct GearsDirectoryImpl* gearsDirectory, struct
        Integer* name, __code next(...)) {
3       struct FTree* cDirectory = new FTree();
4       cDirectory = gearsDirectory->currentDirectory;
5       struct Node* node = new Node();
6       node->key = name->value;
7       goto cDirectory->get(node, cd2Child2);
8   }
9
10  __code cd2Child2(struct GearsDirectoryImpl* gearsDirectory, struct
        Node* node, __code next(...)) {
11      struct FTree* iNodeTree = new FTree();
12      iNodeTree = gearsDirectory->iNodeTree;
13      goto iNodeTree->get(node->value, cd2Child3);
14  }
```

```
15
16 __code cd2Child3(struct GearsDirectoryImpl* gearsDirectory, struct
       Node* node, __code next(...)) {
17     gearsDirectory->currentDirectory = node->value;
18     goto next(...);
19 }
```

5.4　非破壊的編集による Backup

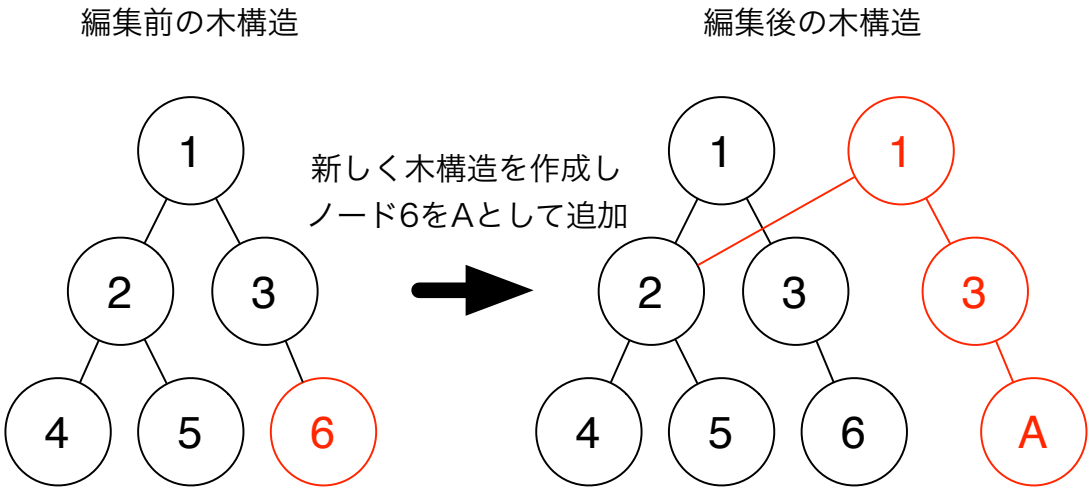

図 5.1　非破壊的な Tree 編集

第 6 章

File 構造

ファイルシステムはディレクトリ構造の他にファイル構造を持つ．GearsOS におけるファイル構造を説明する．

6.1 I/O Stream

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．3 つの Queue を通過するデータは element と呼ばれる．

6.2 log によるバージョン管理

第 7 章

WordCount

WordCount 例題 [?] は GearsOS のファイルシステムを構築する際に用いてる例題である．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 7.1 は WordCount 例題の処理の流れを示している．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．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．

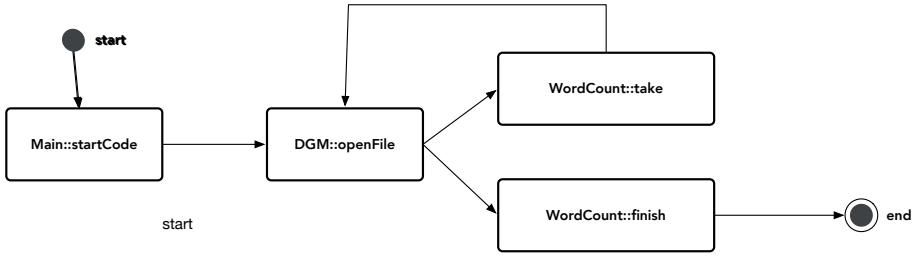

図 7.1　WordCount with CbC

7.1　API

7.2　GearBox 的な処理

第 8 章

今後の課題

8.1　分散ファイルシステム

8.2　信頼性

8.3　shell

第 9 章

まとめ

謝辞