

# Gears OS のファイルシステムとDB

又吉 雄斗<sup>a)</sup> 佐野 巧曜<sup>b)</sup> 河野 真治<sup>c)</sup>

**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。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。DB は SQL によってデータの挿入や変更が可能だがスキーマを事前に定義し、insert 時にそれらの schema を指定する必要がある。GearsOS のファイルシステムでは SQL の機能に相当する grep や find などのインターフェースを実装し、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。

## 1. GearsOS におけるファイルシステムとDB

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。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。

GearsOS へファイルシステムとDBを実装するにあたり、RedBlackTree を用いる。DB はファイルシステムと本質的に同じ役割を持っているため同じ RedBlackTree で表現することが可能であると考えられる。よって、今回は GearsOS におけるファイルシステムとDBを RedBlackTree で実装するための設計を行う。

## 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。

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

<sup>1</sup> 情報処理学会

IPSJ, Chiyoda, Tokyo 101-0062, Japan

<sup>†1</sup> 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

<sup>a)</sup> matac@cr.ie.u-ryukyu.ac.jp

<sup>b)</sup> yoshisaur@cr.ie.u-ryukyu.ac.jp

<sup>c)</sup> kono@ie.u-ryukyu.ac.jp

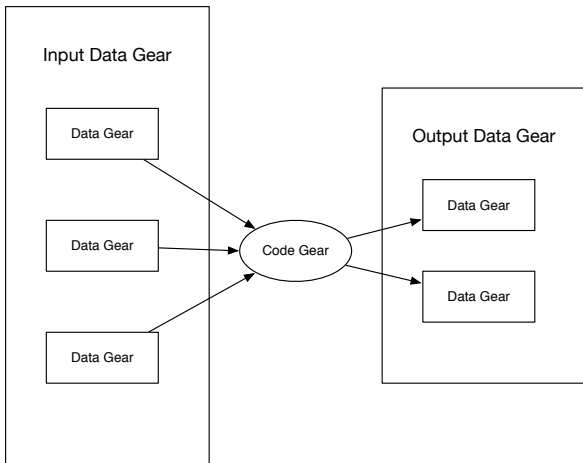


図 1: CodeGear と入出力の関係図

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 *argv[]) {
    int n = 1;
    goto add1(n);
}

```

### 3. 信頼性の保証を目的とした 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 は現在開発途中であり、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、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。

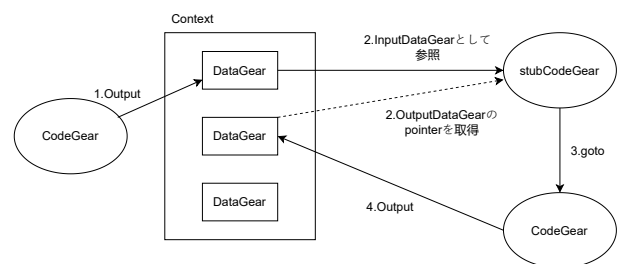


図 2: Context を参照する流れ

図 3 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。

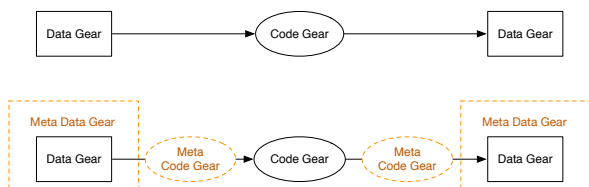


図 3: CodeGear と MetaCodeGear の関係

GearsOS には現在 3 つの種類がある。1 つ目が型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。これは、Agda によって実装されている。2 つ目が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、RedBlackTree でのディレクトリシステムの構築するなどの取り組みもされている [7]。3 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 [8]。これは、教育用に開発された x.v6 [9] を CbC で書き換える形で実装する。今回、ファイルシステムを実装する対象は 3 つ目の CbC\_xv6 である。

#### 4. RedBlackTree によるファイルシステムの構成

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 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。

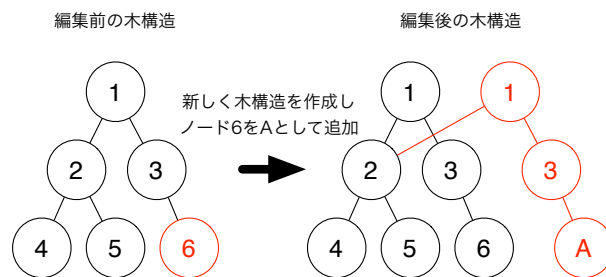


図 4: 非破壊的な Tree 編集

#### 5. ディスク上とメモリ上のデータ構造

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。

#### 6. データのロールバックとバックアップ

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。このような、ロールバックとバックアップの仕組みをファイルシステムに実装したい。

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 編集はアップデートのたびに、ルートノードを増やす。つまり、ルートノードはアップデートのログと言えその時点のデータのバージョンを表していると考えられる。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。

## 7. 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書き換える。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 5 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。

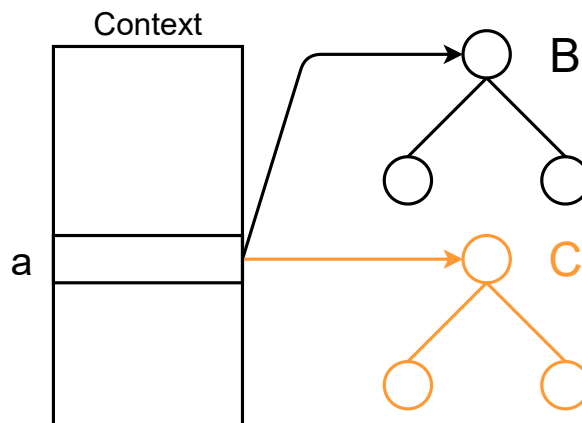
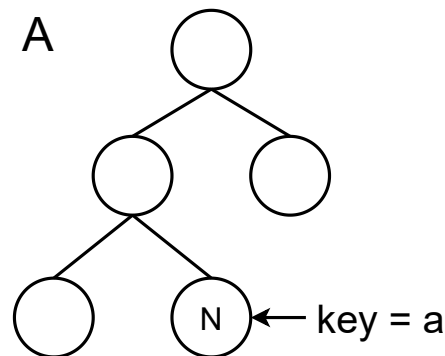


図 5: トランザクショナルな write 操作

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。

## 8. 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。

## 9. 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。

## 10. 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。

### 10.1 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。

### 10.2 ログなどの時系列データの保存

時系列データは通常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。

## 10.3 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。

## 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] 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 (2022).
- [8] 並列信頼研究室: CbC\_xv6, [http://www.cr.ie.u-ryukyu.ac.jp/hg/CbC/CbC\\_xv6/](http://www.cr.ie.u-ryukyu.ac.jp/hg/CbC/CbC_xv6/).
- [9] 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>.
- [10] 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).
- [11] 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).
- [12] 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).
- [13] 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).
- [14] 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).