GearsOSの分散ファイルシステムの設計

一木貴裕[,a)]　河野 真治[,b)]

概要：本研究室では gear というプログラミング概念を持つ, 分散フレームワーク Christie を開発している. Christie はノード同士が Datagear と呼ばれる変数データを送信しあうことにより, 簡潔に分散プログラムの記述を行うことができる. この Christie の仕組みを, 同様に本研究室が開発している GearsOS に組み込み, ファイルシステムを構築したい. GearsOS はノーマルレベルとメタレベルを分けて記述できる Continuation based C(CbC) で構成されており、Christie と近い仕様をもつ.

Designing a Distributed File System for GearsOS

1. GearsOS のファイルシステムの開発

当研究室では OS の信頼性の検証を目的とした OS である GearsOS を開発している. GearsOS はユーザレベルとメタレベルを分離して記述が行える言語である Continuation based C(以下 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.

GearsOS は現在開発途上であるため, 現在は言語フレームワークとしてしか動作しない。OS として起動するためにこれから実装が必要な機能が多く存在しており, その中の一つとして分散ファイルシステムが挙げられる. GearsOS の分散ファイルシステムを構成するために、当研究室が開発している分散フレームワーク Christie の仕組みを用いようと考えた.

Chrsitie は GearsOS のもつ Gear という概念とよく似た, 別の Gear というプログラミング概念を持っており, DataGear と呼ばれる変数データを接続されたノード同士が送信しあうことで分散処理を簡潔に記述することができる. DataGear は指定された型と名前を持つ key に対応しており, プログラムが必要な key にデータが揃ってから初めてプログラムが処理される. また, Chrisite は Topology-Manager と呼ばれる機能を持っており, 任意の形でノード同士の配線を行い Topology を形成する機能を持っている.

1　情報処理学会
IPSJ, Chiyoda, Tokyo 101–0062, Japan
†1　現在，琉球大学理工学研究科情報工学専攻
Presently with Johoshori Uniersity
a)　ikki-tkhr@cr.ie.u-ryukyu.ac.jp
b)　kono@ie.u-ryukyu.ac.jp

GearsOS の分散処理の記述方式として Christie の仕組みを取り入れることにより, 分散ファイルシステムを構成したい.

2. UNIX ファイルシステムについて

UNIX ファイルシステムは UNIX 系 OS を始めとした多くの OS のファイルシステムの大元となっている.

UNIX ファイルシステムではファイルに構造を持たず, カーネルがファイルを単純なバイト配列とみなして処理が行われ, ファイルの実際の取り扱いはプログラムが判断する. ユーザの視点ではファイルを取り扱う際に, OS レベルの観点からファイルを構成する必要がなく, ファイルの拡張子を任意のアプリケーションに適したものにするだけで良い. 全てのファイルは i-node と呼ばれるユニークな番号をもつデータブロックで表される, ファイルのメタデータは全て i-node 内に格納されている. ディレクトリ を格納する i-node はファイルの探索の際などに用いられ, ユーザは stat() と呼ばれるシステムコール により大きさなどのファイルのメタ情報を得ることができる.

また, UNIX はデバイスにまたがる仮想ファイルシステムを搭載している. 例えばネットワークを経由してファイルシステムにアクセスする際は, 木構造に構成されたディレクトリ Tree にリモート先のファイルシステムをマウントすることにより参照することができる.

UNIXbase のファイルシステムを比較・検討をしながら GearsOS を構成していく.

© 1959 Information Processing Society of Japan

3. Continuation based C

GearsOS はC言語の下位言語である Continuation based C を用いて記述されている．CbC は関数呼び出しでなく，継続を導入しており，スタック領域を用いず jmp 命令でコード間を移動することにより軽量な継続を実現している．CbC ではこの継続を用いて for 文などのループの代わりに再起呼び出しを行う．実際の OS やアプリケーションを記述する際には GCC または LLVM/clang の CbC 実装を用いる．

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` で宣言を行い，各 CodeGear は DataGear と呼ばれる変数データを入力として受け取り，その結果を別の DataGear に書き込む．特に入力の DataGear を InputDataGear，出力される DataGear を OutputDataGear と呼ぶ．CodeGear と DataGear の関係図を図 1 に示す．CodeGear は関数呼び出しのスタックを持たないため，一度 CodeGear を遷移すると元の処理に戻ってくることができない．

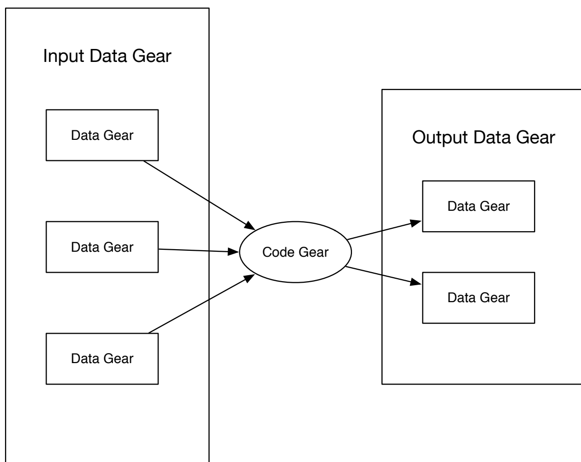


図 1: CodeGear と入出力の関係図

CbC コードの例をソースコード 1 に示す．この例題では特定のシステムコールの場合，CbC で実装された処理に goto 文をつかって継続する．例題では CodeGear へのアドレスが配列 cbccodes に格納されている．引数として渡している `cbc_ret` は，システムコールの返り値の数値をレジスタへ代入する CodeGear である．実際に `cbc_ret` に継続が行われるのは， read などのシステムコールの一連の処理の継続が終わったタイミングである．

Code 1: CbC を利用したシステムコールのディスパッチ

```
void syscall(void)
{
    int num;
    int ret;

    if((num >= NELEM(syscalls)) && (num <= NELEM(
        cbccodes)) && cbccodes[num]) {
        proc->cbc_arg.cbc_console_arg.num = num;
        goto (cbccodes[num])(cbc_ret);
    }
```

Code1 の状態遷移図を図 1 に示す。図中の `cbc_read` などは， `read` システムコールを実装している CodeGear の集合である。

4. CbC を用いた OS の記述

CodeGear の遷移はノーマルレベルから見ると単純に CodeGear が DataGear を Input, Output を繰り返し，コードブロックを移動しているように見える．CodeGear が別の DataGear に遷移する際の DataGear との関係性を図 2 に示す．ノーマルレベルでは DataGear を受け取った CodeGear を実行，実行結果を DataGear に書き込み別の CodeGear に継続していると見える．

しかし，実際には CodeGear から別の CodeGear への遷移にはデータの整合性の確認などのメタ計算が必要となる．コード間の遷移に必要となるメタ計算は，MetaCodeGear と呼ばれる CodeGear ごとに実装された CodeGear で行う．MetaCodeGear で参照される DataGear を MetaDataGear と呼び，また，CodeGear の直前に実行される MetaCodeGear を StubCodeGear と呼ぶ．これら Meta 計算部分を含めた CodeGear の遷移と DataGear の関係性を図示すると図 2 の下段の形に表せる．CordGear の実行前後に実行される MetaCodeGear や入出力の DataGear を MetaDataGear から取り出すなどのメタ計算が加わる．

MetaCodeGear は詳細な処理の変更や，スクリプトに問題がある場合を除き，プログラマが直接実装する必要がなく，GearsOS が持つ Perl スクリプトにより，GearsOS がビルドされる際に生成される．

CodeGear の遷移に重要な役割を持つ MetaDataGear として context が存在する。context は遷移先の CodeGear と MetaDataGear の紐付けや，計算に必要な DataGear の保存や管理を行う．加えて context は処理に必要になる CodeGear の番号と MetaCodeGear の対応表や，DataGear の格納場所を持つ．context と各データ構造の役割を図 3 に示す．計算に必要なデータ構造と処理を持つデータ構造であることから，context は従来の OS のプロセスに相当し，ユーザープログラムごとに context が存在している．

5. 分散フレームワーク Christie

Christie は当研究室で開発されている java 言語で記述された，分散フレームワークである．Christie は CbC と似ているが異なる仕様を持つ Gear というプログラミング概念を持つ．

© 1959 Information Processing Society of Japan

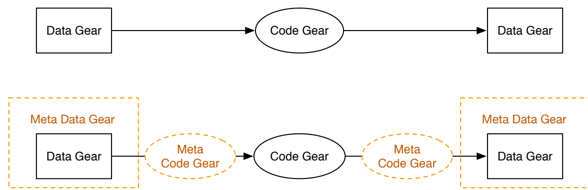
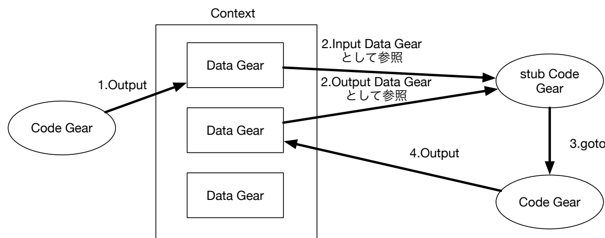


図 2: CodeGear と MetaCodeGear の関係図

図 3: Context を介した CodeGear の継続



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

CodeGear はクラスやスレッドに相当する．DataGear は変数データであり，CodeGear 内で java のアノテーションを用いて記述する．DataGear は Key と必ず対応しており，put と言う処理により CG 内の全ての Key に DataGear が揃った際に初めて CG が動作するという仕組みになっている．

CodeGearManager はいわゆるノードに相当し，CodeGear, DataGear, DataGearManager を管理する．複数の CodeGearManager 同士が配線され，DG を送信し合うことで分散処理を実現している．

DataGearManager は DG を管理しているもので変数プールに相当し，CGM が利用する CG の key と put されたデータの組み合わせを所持している．DataGearManager は Local と Remote に区分することができ，LocalDGM は CGM 自身が所持する DG のプールであり，Local に put することにより自身の持つ key に DG を送ることができる．対する RemoteDataGearManager は CGM が配線されている別の CGM が持つ DG のプールである．つまり，任意の接続された RemoteDGM に DG を put すると，対応した CGM(ノード) が持つ DGM の pool に DG が送信される．RemoteDGM に DG を put する処理が分散処理の肝となっている．RemoteDataGearManager の仕組みを図 4 に示す．

Christie の要となる DataGear の key は java のアノテーション機能が使われている．アノテーションには以下の 4 つが存在する．

Take 先頭の DG を読み込み，その DG を削除する．DG が複数ある場合，この動作を用いる．

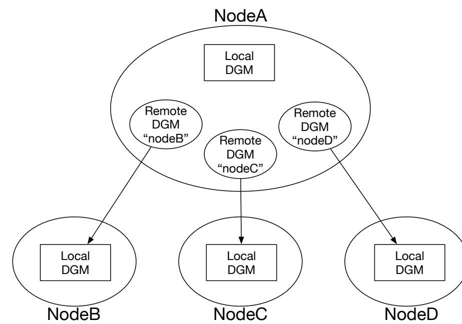


図 4: RemoteDataGear と接続ノードの関係図

Peek 先頭の DG を読み込むが，DG が削除されない．そのため，特に操作をしない場合は同じデータを参照し続ける．

TakeFrom(Remote DGM name) Take と似ているが，Remote DGM name を指定することで，その接続先 (Remote) の DGM から Take 操作を行える．

PeekFrom(Remote DGM name) Peek と似ているが，Remote DGM name を指定することで，その接続先 (Remote) の DGM から Peek 操作を行える．

5.1 Christie のサンプルコード

コード 2, 3 は Christie で記述した Hello World のプログラムである．ユーザープログラムは StartCodeGear クラスを継承したクラス (コード 2) から開始する．CodeGearManager はポート番号を指定した上で createCGM メソッドを呼び出すことにより生成される．生成された CodeGearManager は CGM 名.setup にて CGM に処理させたいスレッド，つまり CodeGear を持たせることができる．

コード 3 は HeloWorldCodeGear の記述である．HeloWorldCodeGear では key: helloWorld に put された文字列を print 出力するという単純な処理を記述している．CGM 名.getLocalDGM().put(”Keyname”, 変数データ) にて key に変数データを紐付け (put し), CodeGear に設定されている全ての key がデータを受け取った際に初めて CodeGear は処理される．HelloWorldCodeGear では String 型の helloWorld という key が Take 型で設定されている．

Code 2: Christie における CGM と CG の setup

```
public class StartHelloWorld extends StartCodeGear {

    public StartHelloWorld(CodeGearManager cgm) {
        super(cgm);
    }

    public static void main(String[] args){
        CodeGearManager cgm = createCGM(10000);
        cgm.setup(new HelloWorldCodeGear());
        cgm.getLocalDGM().put("helloWorld","hello");
```

© 1959 Information Processing Society of Japan

```
        cgm.getLocalDGM().put("helloWorld","world");
    }
}
```

Code 3: HelloWorldCodeGear

```
public class HelloWorldCodeGear extends CodeGear {
    @Take
    String helloWorld;

    @Override
    protected void run(CodeGearManager cgm) {
        System.out.print(helloWorld + "␣");
        cgm.setup(new HelloWorldCodeGear());
    }
}
```

5.2 TopologyManager

Christie には Topology を形成するための機能 Topology-Manager が備わっている．Topology に参加するノードに対して名前を与え, 必要とあればノード間の配線を行う．

TopologyManager の Topology 形成方法として静的 Topology と動的 Topology がある．静的 Topology はプログラマが任意の形の Topology とノードの配線を dot ファイルに記述し, TopologyManager に参照させせることで自由な形の Topology が形成できる．現時点では静的 Topology での Topology 形成は dot ファイルに記述した参加ノード数に実際に参加するノードの数が達していない場合, 動作しないという制約が存在している．動的 Topology は参加を表明したノードに対し, 自動的にノード同士の配線を行う．例えば Tree を構成する場合, 参加したノードから順番に root から近い役割を与える．

6. GearOS のファイルアクセス

Christie の分散ファイルシステムを WordCount 例題を通して構築する．WordCount 例題とは指定したファイルの中身の文字列を読み取り, 文字数と行列数, 加えて文字列を出力すると言う例題である．CbC で記述した場合の WordCount プログラムの遷移図を示した図を図に示す．WordCount の例題は大きく分けて, 指定した名前のファイルを FIle 構造体として Open する FileOpen スレッド, ファイル構造体を受け取り文字列 (word) を表示し文字数 (bytes) と行数 (lines) 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．図 5 で示した WordCount の遷移図は File をオープンした CG と WordCount の CG を巡回することにより, ブロックを処理していく．

WordCount とファイルの接続は UNIX のシェルのようにプログラムの外で接続される．この接続は Topology-Manager によって接続される．

ファイルシステム側では Block を接続された CG へ goto で接続すればよい．ここでは UNIX 上の GearsOS であるので UNIX のファイルシステム API を経由して Block を投げる．この記述ではファイルの読み込みと WordCount の処理がコードとデータの流れに沿って起こる．

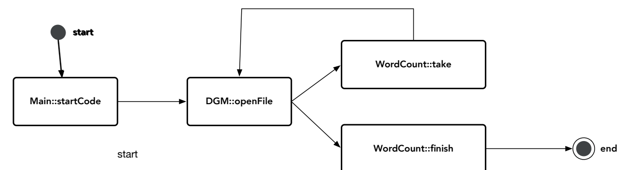


図 5: WordCount with CbC

7. ChristieAPI

GearsOS 上のファイルは名前のついた大域的な資源であり, 複数のプロセスから競合的にアクセスされる．前節の API ではファイルと WordCount が直接的に結合されているので競合的なアクセスは起きない．そこで DataGears Stream に名前をつけてアクセスする API を導入する．これは java version の Christie の GearsOS 版となる．

図 6 は File と分離された WordCount の結合を表している．File Interface は Block を WordCount に送信するが, 直接的に送るのではなく, WordCount の名前がついた DataGearManager の ploxy に書き込む．FileInterface からは RemoteDGM として見れる．RemoteDGM に書き込む方法は二通りあり, 一つは goto 文の OutPut に指定する方法がある．この方法は従来の API をつかった方法とほぼ同じになる．もう一つは RemoteDGM へ put する API を使う．この手法では複数の RemoteDGM に書き込むことができる．

DG の受け取りは CodeGear の入力で行われる．入力の接続は TopologyManager に行われるが, 入力の DG に key を設定する metaCG を使う．他の CG が put した DG に対応する CG がその context で実行される．実行された CG は DG を DGM に書き込むことで計算が進む．

同じ名前 (key) を持つ DG は複数の CG から競合的にアクセスされる．RemoteDGM は LocalDGM の ploxy であり, 他の計算ノードにあって良い．DGM の同期は GearsOS が管理する．

File の eof などは DG にフラグをつけて関数型プログラミング的に処理される．

図 6 は WordCount を RemoteDGM を用いて記述した際の遷移図である．NodeA にて任意のファイルを開き DGM を通して 1 行ごとに文字列を NodeB に送信 (put), NodeB にて WordCount の処理を行い, Output として NodeB から WordCount の処理を送り返す．NodeA 側から送信するファイルの行がなくなった場合, NodeB 側に eof を送信して双方の処理を終了するといった流れになる．

© 1959 Information Processing Society of Japan

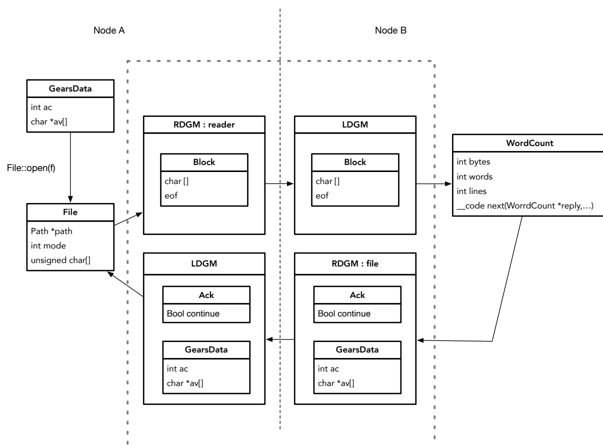


図 6: WordCountDGM with Christie

Code 4: UnixFileImpl.cbc

```
#include <stdio.h>
#impl UnixFileImp as "UnixFileImpl.h"

File* createUnixFileImpl(struct Context* context) {
    File *file = new File();
    file->FileImpl = (union Data*)new FileImpl();
    return file;
}

readBlock(UnixFileImpl* file) {
   Block *block = new Block();
   int len = read(fd, BUFSIZE, block->data);
   block->eof &= ~BLOCK_FLAG_EOF;
   if (len <=0 ) {
       block->eof |= BLOCK_FLAG_EOF;
       close(file->fd);
   }
   return block ;
}

__code unixOpen(UnixFileImpl* file,Key *key, __code
     next(Block *block,...));
   file->fd = open(key->path,unix_mode(key->modde));
   if (fd < 0) {
       goto error("can't␣open");
   }
   goto next(readBlock(file), ...);
}

__code uniAck(UnixFileImpl* file,Ack *ack, __code
     next(Block *block,...));
   if (!ack->isOk) {
      close(file->fd);
      goto next(...);
   }
   goto next(readBlock(file), ...); // file is
        automaticaly put into local dataGearManger/
        input
}
```

Code 5: WcImpl.cbc

```
#include "../../../context.h"
#include <stdio.h>
#impl "Wc.h" as "WcImpl.h"
#interface "WcResult.h"

Wc* createWcImpl(struct Context* context) {
    Wc *wc = new Wc();
    wc->wc = (union Data*)new WcImpl();
    wc->bytes = 0;
    wc->words = 0;
    wc->lines = 0;
    return wc;
}

__code take(Impl* wc, Block *block,__code next(Ack *
     ack, ...),__code finish(StdData *result,...) {
    if (isEof(block->eof )) {
        result.buffer = new Buffer(1);
        result.buffer->data = new Byte(BUSIZE);
        result.size = 1;
        result.buffer->size =
            snprintf(result.buffer[0]->data, "%d␣%d␣%d
                 \n",wc->bytes,wc->words,wc->lines);
        goto finish(resut);
    }
    for(size_t i = 0 ; i<block->size; i++) {
        if (block->data[i] == '\n') wc->lines++;
        if (block->data[i] == '␣') {
            wc->words++;
            while(block->data[i] == '␣') {
               if(i>=block->size)
                    goto next(ack,take);
               i++;
               wc->bytes++;
            }
        }
        wc->bytes++;
    }
    goto next(ack,take);
}
```

8. FileSystem Implementation

GearsOS 独自のファイルシステムは単なる DG のリストとなる．要求された DG をリストから参照して goto または put で送信する．acknowledge を待って順次送信すれば良い．この場合では DG はメモリ上にのみ存在する．

持続的なファイルシステムの場合はリストまたは B-Tree を SSD や MVME などの持続性のあるデバイスに格納する．これらのデバイスは単なるメモリとして扱ってよく，メモリ上のデータ構造と同様に構築する．

メモリ領域の管理はメタ計算として実装される．メタレベルで GC や DataGear Pool を用いて不要になったメモリの回収を行う．

このように DGM 自体がファイルの概念に対応する．DGM に格納されている DG には型があり，正しく接続するには型変換を提供する必要がある．UNIX 的に std Data のようなものを提供しても良い．DGM のデータ構造は複数のものが可能になり，Red-Black-Tree あるいは単方向リストあるいは双方向リストを使うことができる．物理デバ

© 1959 Information Processing Society of Japan

イス上の重複管理や変更履歴管理も DGM が行う．しかし，ノーマルレベル的には単なる DG のリストとして扱う．つまり検証上は単なるリストとなる．図 7 はファイルを別ノードにて参照する際の遷移図である．WordCount と同様な 1 行ごとの文字列に加え, 読み込みモード, ファイルのパスなどを put する. 送信の制御として Ack もデータとして送信している．図 8 はデバイス上に保存された DGM を LocalDGM として呼び出し, 操作を行う際の遷移図となる.

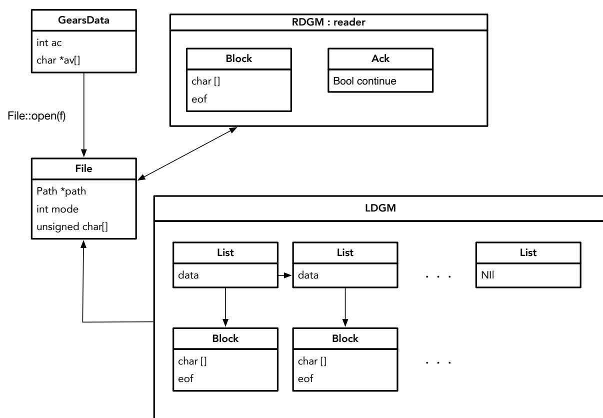


図 7: file Implementation

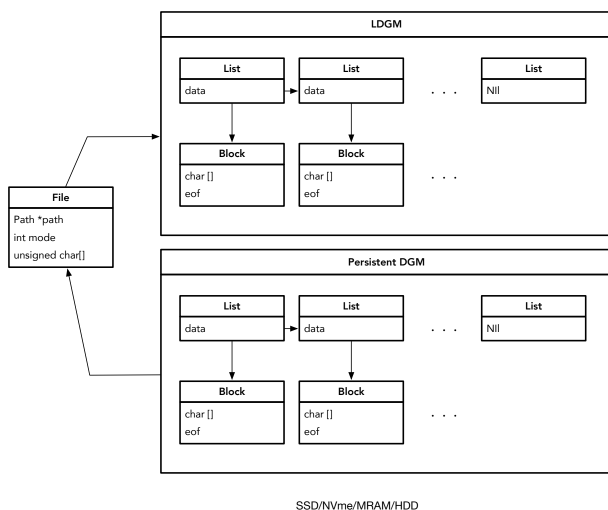


図 8: file Persistency

9. 競合的アクセスを含む分散計算の検証

ChristieAPI は競合的なアクセスを含むので逐次型プログラミングのように検証することができない．Topology-Manager は名前付き DataGearManager を相互に接続するが, これも動的に変更される．これ全体をモデル検査することを考える．可能な接続パターンと可能な DGM の内容のパターンを網羅すれば良い．一般的にはこれは無限あるいは膨大となるが, 抽象化を導入することにより, より様々なレベルでの検証が可能になることが期待される.

接続と DGM の内容のパターンが確定すれば, その範囲でプログラムは関数型プログラミングとして振る舞う．この場合の検証は Hoare Logic などで証明的に行うことができる．証明が複雑な場合でも, DG のパターンをメタ計算で調べるなどの手法を用いることができる.

10. 比較とまとめ

GearsOS におけるファイルシステム API の設計に対して議論を行った．API は二種類あり, アプリケーションで閉じた決定的な実行を行う直接接続されたものと, もう一つは DataGearManager に名前をつけて競合的アクセスを許す形でゆるく接続されるものである.

DataGearManager はファイルとして見ることもできるし, 分散環境での通信として見ることもできる．ファイルシステムの同期, 例えば DataGear の待ち合わせは Synclonised Queue 的に行われる．Peek を用いるとブロックしない形で Read Only アクセスすることが可能になる．いずれの場合もアクセスは DataGear 単位であり, 不完全な状態なデータになることはない.

UNIX FileSystem は API 的には File Stream と Socket Stream は Read-Write でアクセスする．しかし, その設定はプログラム内部で煩雑な処理が必要となる．GearsOS ではこの部分は TopologyManager に押し付けられている. UNIX では Stream に型がないので不完全なデータが生じてしまう．UNIX ファイルシステムには fsck と呼ばれる修復機能があるが, メモリに対する修復機能は存在しない. GearsOS 側ではそれらは DGM のメタな機能として実装することが可能である．例えばメモリの一部不良などに対応する DGM を作るといったことが可能になると思われる.

DGM の名前とその中の DataGear Stream に対応する key でアクセスする対象が決まる．これが UNIX でいう i-node 番号に相当する．Human Readable な名前空間はそれらに対して自由に構築して良い．UNIX のファイルシステムと異なり, i-node の構成と名前空間の構成は独立で良い．例えば scrap box のような Tag base でのアクセスが考えられる.

参考文献

[1] 赤堀 貴一，河野真治：**Christie によるブロックチェーンの実装**．琉球大学工学部情報工学科卒業論文 2019.
[2] 照屋 のぞみ，河野真治：**分散フレームワーク Christie の設計**．琉球大学理工学研究科修士論文 2018.
[3] 清水 隆博, 河野真治：**xv6 の構成要素の継続の分析**．OS 研究会 2019.
[4] 清水 隆博，河野真治：**継続を基本とした言語による OS のモジュール化**．琉球大学理工学研究科修士論文 2020.
[5] 宮城 光輝，河野真治：**継続を基本とした言語による OS のモジュール化**．琉球大学理工学研究科修士論文 2020.