令和元年度 卒業論文

CbCによるxv6のFileSystemの書き換え

琉球大学工学部情報工学科

165723C 坂本昂弘

指導教員 河野真治

目 次

図 目 次

ソースコード目次

第1章　xv6 の OS の信頼性保証

第2章　Continuation based C

2.1　Continuation based C の概要

Continuation based C [1] (以下 CbC) は基本的な処理単位を CodeGear として定義し、CodeGea 間で遷移するようにプログラムを記述する C 言語と互換性のある当研究室で開発されたプログラミング言語である。図 2.1 は CodeGear 間の継続する際の処理の流れを示している。

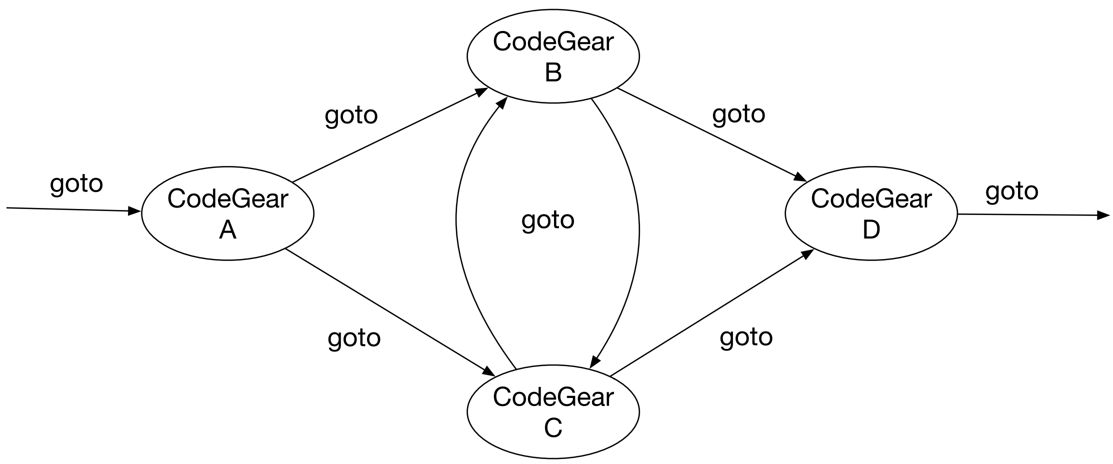

図 2.1: CodeGear 間の継続

また、プログラムを記述する際、ノーマルレベルの記述と別にメモリ管理、スレッド管理など記述しなければならない処理が存在する。これらをメタ計算と呼ぶ。CbC の CodeGear 、DataGear にはそれぞれ Meta CodeGear 、Meta DataGear と呼ばれるメタレベルの単位が存在する。これらを用いることによってメタ計算を実現し、 OS の信頼性を保証できる。

現在 CbC は C コンパイラである GCC[2] [3] 及び LLVM[4] [5] をバックエンドとした clang 上で実装されている。本研究では、このプログラミング言語を用いて xv6 の Filesystem を書き換える。

2.2　CodeGear

CodeGear は CbC における基本的な処理単位である。以下のソースコード 2.1 は CodeGear の継続の例である。

ソースコード 2.1: CodeGear の継続の例

```
1 __code cg0(Integer a, Integer b){
2  int a_v = a->value;
3  int b_v = b->value;
4  Integer c = {a_v + b_v};
5   goto cg1(c);
6 }
7 __code cg1(Integer c){
8   goto cg2(c);
9 }
```

CodeGear は__code CodeGear 名 (引数) の形で記述される。CodeGear は返り値を持たない為、関数内で処理が終了すると呼び出し元の関数に戻ることがなく別の CodeGear へ遷移する。ソースコード 2.1 の 5 行目の goto cg1(c); や 8 行目の goto cg2(c); などがこれにあたる。図 2.2 はソースコード 2.1 の状態遷移を表している。

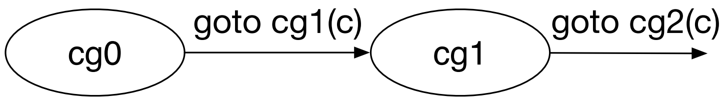

図 2.2: ソースコード 2.1 が表している CodeGear の状態遷移

また CbC における CodeGear 間の継続にはスタックが使用されず、呼び出し元の環境などを持たない為軽量継続と呼ぶ。この CbC における CodeGear 間の継続にスタックが使用されない性質は信頼性の高い OS の開発に適している。

2.3 DataGear

DataGear は CbC におけるデータの基本的な単位である。CodeGear は Input DataGear、Output DataGear を引数に持ち、図 2.3 で示すように遷移する際に任意の Input DataGear を参照し、Output DataGear を書き出す。

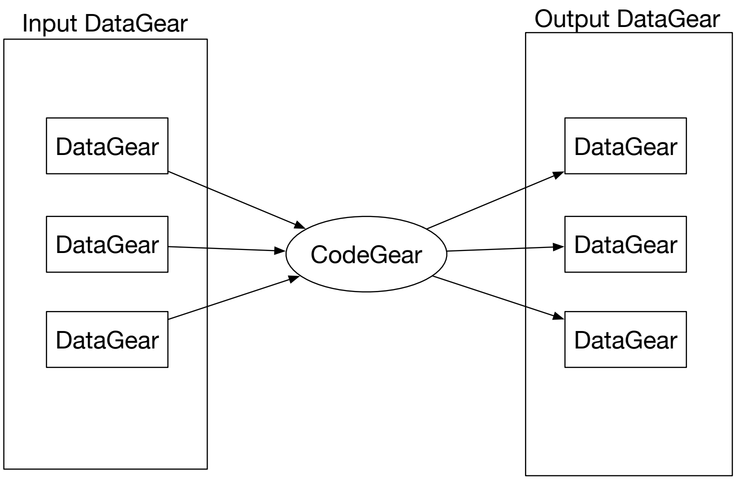

図 2.3: CodeGear と DataGear

2.4 Meta CodeGear と Meta DataGear

Meta Code Gear は図 2.4 で示すようにノーマルレベルの Code Gear の直後に遷移され、メタ計算を実行する。

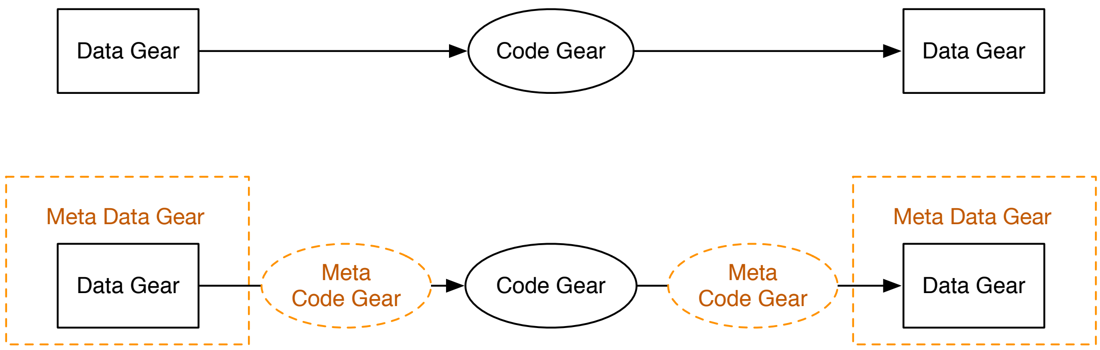

図 2.4: メタレベルの処理を可視化した CodeGear と DataGear

Meta Data Gear は CbC の 接続可能な Code Gear、 Data Gear のリストであり Data Gear を確保するメモリ空間でもある。ノーマルレベルからの書き換えやアクセスを防ぐために存在している。GearsOS の Context がこれにあたる。

第3章　GearsOS

3.1　GearsOS の概要

Gears OS [6] は CbC によって記述されており、CodeGear と DataGear の単位を用いて開発されている OS である。Gears OS は 一連の実行が行われる際に使用される CodeGear と DataGear を全て持つ Context と呼ばれるものを持っている。Gears OS は CodeGear 間の継続などの際、常に context を持ち歩いており CodeGear と DataGear の参照が必要になる場合、この Context を通して参照される。

3.2　Context

Context とは一連の実行が行われる際に使用される CodeGear と DataGear の集合である。 従来のスレッドやプロセスに対応する。Context は接続可能な CodeGear、 Data Gear のリスト、 Data Gear を確保するメモリ空間、 実行される Task への Code Gear 等を持っている。 CodeGear が 別の CodeGear に遷移する際、 必ず context を参照し enum で定義された CodeGear の番号を指定し遷移する。ノーマルレベルで見た際の CodeGear、DataGear および context の関係を以下の図 3.1 に簡潔に示す。

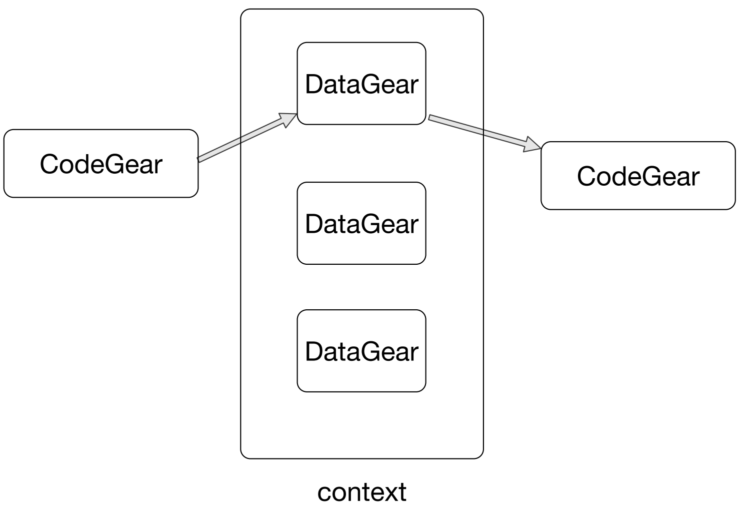

図 3.1: CodeGear、DataGear、contxt の関係図

3.3 Inetrface

Interface は Gears OS のモジュール化の仕組みである。 Interface は呼び出しの引数になる Data Gear の集合であり、そこで呼び出される Code Gear のエントリである。呼び出される Code Gear の引数となる Data Gear はここで全て定義される。Interface を定義することで複数の実装を持つことができる。この Interface は、 Java の Interface や Haskell の型クラスに対応し、導入することで仕様と実装に分けて記述することが出来る。図 3.2 は Stack の Interface とその実装を表したものである。

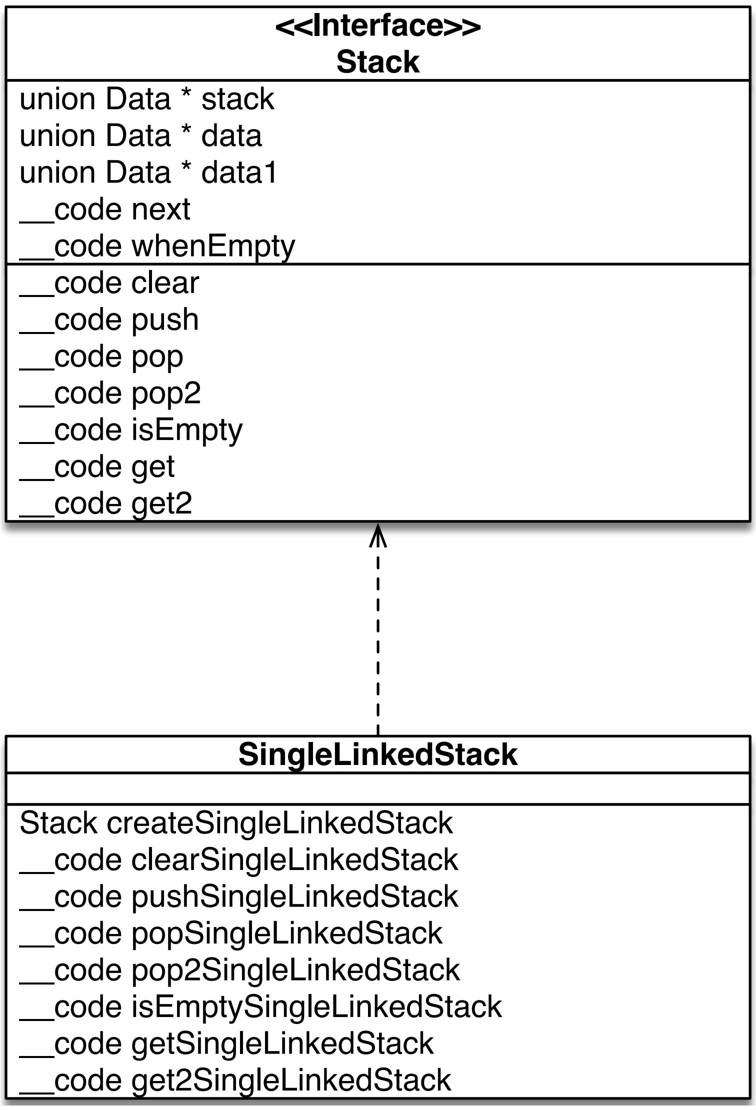

図 3.2: Stack の Interface とその実装

第4章　xv6

4.1　xv6 の概要

xv6 [7] とは MIT のオペレーティングコースの教育目的で 2006 年に開発されたオペレーティングシステムである。xv6 はオリジナルである v6 が非常に古い C 言語で書かれてい る為、ANSI-C に書き換えられ x86 に再実装された。 xv6 は read や write などの systemcall、 プロセス、 仮想メモリ、 カーネルとユーザーの分離、 割り込み、 ファイルシステムなど Unix の基本的な構造を持っている。本研究で使われているのは ARM[8] 上で動作する Raspberry Pi 用に改良された xv6 を使用する。

4.2　xv6 の FileSystem

FileSystem とは、コンピュータの資源を操作するための OS が持つ機能のことである。ファイルといえば記憶装置内に格納されている情報を指すが、xv6 の FileSystem は、デバイスやプロセス、カーネル内の処理をする際の情報などをファイルとして扱う。OS ごとに利用している FileSystem は異なるが、一部の OS を除きほとんどの OS には FileSystem が存在する。xv6 の FileSystem は図 4.1 のように 7 つの階層によって構成されている。

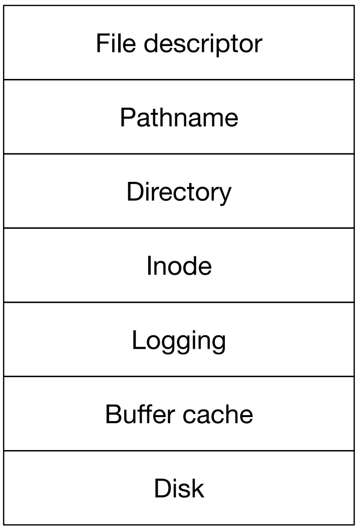

図 4.1: xv6 の FileSystem Layer

4.3 FilleSystem の API

FileSystem について記述している fs.c ではファイルを操作、管理する際に様々な関数がプロセスやデバイスなどから呼び出され使用されている。fs.c に存在している関数とその挙動に関して具体的に以下に示す。

- readsb
  ブロックのファイルサイズやデータブロックの数、inode の数、log 中のブロック数などが格納されている super block を読み込む。

- iinit

- ialloc
  デバイスで指定されたタイプを新しい inode に割り当てる。

- iupdate
  変更されたメモリ内の inode をディスクにコピーする。

- idup
  IP の参照カウントをインクリメントする。

- ilock
  指定した inode をロックする。またその際に必要であるならば、ディスクから inode を読み込む。

- iunlock
  指定された inode のロックを解除する。
- iput
  メモリ内の inode への参照を削除する。
- iunlockput
  指定された inode のロックを解除してから iput を実行する。
- stati
  inode から ファイルに関する統計情報を複製する。
- readi
  inode からデータを読み込む。
- writei
  inode へデータを書き込む。
- namecmp
- dirlookup
  ディレクトリ内のディレクトリエントリを探す。
- dirlink
  新しいディレクトリエントリ（名前、inum）をディレクトリ dp に書き込む。
- namei
- nameiparent

第5章　CbCによるFileSystemの書き換え

5.1　書き換え方針

5.2　FileSystem の Interface の定義

インターフェースはある Data Gear の定義と、それに対する操作を行う Code Gear の集合を表現する Meta Data Gear である。Context では全ての Code Gaer と Data Gear の集合を表現していることに対し、インターフェースは一部の Code Gear と一部の Data Gear の集合を表現する。インターフェースを記述することによってノーマルレベルとメタレベルの分離が可能となる。

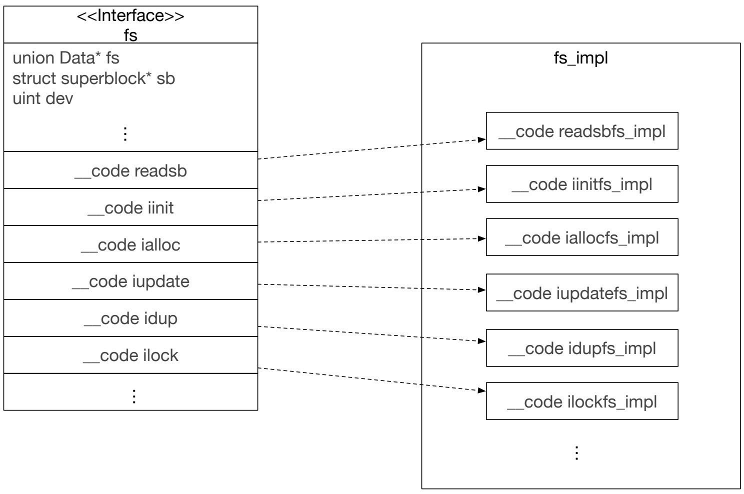

図 5.1: xv6 FileSystem の Interface と実装

FileSystem の Interface を記述したコードをソースコード 5.1 に示す。

ソースコード 5.1: FileSystem の Interface (fs.dg)

```
typedef struct fs<Type,Impl> {
    __code readsb(Impl* fs, uint dev, struct superblock* sb, __code next(...));
    __code iinit(Impl* fs, __code next(...));
    __code ialloc(Impl* fs, uint dev, short type, __code next(...));
    __code iupdate(Impl* fs, struct inode* ip, __code next(...));
    __code idup(Impl* fs, struct inode* ip, __code next(...));
    __code ilock(Impl* fs, struct inode* ip, __code next(...));
```

```
8     __code iunlock(Impl* fs, struct inode* ip, __code next(...));
9     __code iput(Impl* fs, struct inode* ip, __code next(...));
10    __code iunlockput(Impl* fs, struct inode* ip, __code next(...));
11    __code stati(Impl* fs , struct inode* ip, struct stat* st, __code next(...));
12    __code readi(Impl* fs, struct inode* ip, char* dst, uint off, uint tot, uint n,
          __code next(int ret, ...));
13    __code writei(Impl* fs, struct inode* ip, char* src, uint off, uint tot, uint n,
          __code next(int ret, ...));
14    __code namecmp(Impl* fs, const char* s, const char* t, __code next(int
          strncmp_val, ...));
15    __code dirlookup(Impl* fs, struct inode* dp, char* name, uint off, uint* poff,
          dirent* de, __code next(int ret, ...));
16    __code dirlink(struct fs_impl* fs, struct inode* ip, struct dirent* de, struct
          inode* dp, char* name, uint off, uint inum, __code next(...));
17    __code namei(Impl* fs, char* path, __code next(int namex_val, ...));
18    __code nameiparent(Impl* fs, char* path, char* name, __code next(int namex_val,
          ...));
19    __code next(...);
20 } fs;
```

1 行目で Interface 名を定義している。typedef struct の後に Interface 名 (今回は fs) を書く。Code Gear は __code CodeGear 名 (引数) の形で記述する。第一引数である Impl* fs が Code Gear の型、第二引数以降は Interface の実装時に利用する Code Gear が必要とする引数が入る。Code Gaer は 2 行目から 18 行目のように ”__code [Code Gear 名]([引数]) ”で記述する。この引数が input Data Gear になる。

5.3 FileSystem の Interface の実装

ソースコード 5.2: FileSystem の Interface の実装 (fs_impl.cbc 一部抜粋)

```
#interface "Err.h"
#interface "fs.dg"

fs* createfs_impl(struct Context* cbc_context) {
    struct fs* fs = new fs();
    struct fs_impl* fs_impl = new fs_impl();
    fs->fs = (union Data*)fs_impl;
    fs_impl->fs_impl = NULL;
    fs_impl->sb = NULL;
    fs_impl->ret = 0;
    fs_impl->dev = 0;
    fs_impl->type = 0;
    fs_impl->bp = NULL;
    fs_impl->dip = NULL;
    fs_impl->inum = 0;
    fs_impl->dp = NULL;
    fs_impl->name = NULL;
    fs_impl->off = 0;
    fs_impl->poff = NULL;
    fs_impl->de = NULL;
    fs_impl->tot = 0;
    fs_impl->m = 0;
    fs_impl->dst = NULL;
    fs_impl->n = 0;
    fs_impl->src = NULL;
    fs_impl->allocinode = C_allocinode;
    fs_impl->allocinode_loop = C_allocinode_loop;
    fs_impl->allocinode_loopcheck = C_allocinode_loopcheck;
```

```
    fs_impl->allocinode_noloop = C_allocinode_noloop;
    fs_impl->lockinode1 = C_lockinode1;
    fs_impl->lockinode2 = C_lockinode2;
    fs_impl->lockinode_sleepcheck = C_lockinode_sleepcheck;
    fs_impl->iput_check = C_iput_check;
    fs_impl->iput_inode_nolink = C_iput_inode_nolink;
    fs_impl->readi_check_diskinode = C_readi_check_diskinode;
    fs_impl->readi_loopcheck = C_readi_loopcheck;
    fs_impl->readi_loop = C_readi_loop;
    fs_impl->readi_noloop = C_readi_noloop;
    fs_impl->writei_check_diskinode = C_writei_check_diskinode;
    fs_impl->writei_loopcheck = C_writei_loopcheck;
    fs_impl->writei_loop = C_writei_loop;
    fs_impl->writei_noloop = C_writei_noloop;
    fs_impl->dirlookup_loopcheck = C_dirlookup_loopcheck;
    fs_impl->dirlookup_loop = C_dirlookup_loop;
    fs_impl->dirlookup_noloop = C_dirlookup_noloop;
    fs_impl->dirlink_namecheck = C_dirlink_namecheck;
    fs_impl->dirlink_loopcheck = C_dirlink_loopcheck;
    fs_impl->dirlink_loop = C_dirlink_loop;
    fs_impl->dirlink_noloop = C_dirlink_noloop;
    fs->readsb = C_readsbfs_impl;
    fs->iinit = C_iinitfs_impl;
    fs->ialloc = C_iallocfs_impl;
    fs->iupdate = C_iupdatefs_impl;
    fs->idup = C_idupfs_impl;
    fs->ilock = C_ilockfs_impl;
    fs->iunlock = C_iunlockfs_impl;
    fs->iput = C_iputfs_impl;
    fs->iunlockput = C_iunlockputfs_impl;
    fs->stati = C_statifs_impl;
    fs->readi = C_readifs_impl;
    fs->writei = C_writeifs_impl;
    fs->namecmp = C_namecmpfs_impl;
    fs->dirlookup = C_dirlookupfs_impl;
    fs->dirlink = C_dirlinkfs_impl;
    fs->namei = C_nameifs_impl;
    fs->nameiparent = C_nameiparentfs_impl;
    return fs;
}

typedef struct superblock superblock;
__code readsbfs_impl(struct fs_impl* fs, uint dev, struct superblock* sb, __code next(...)) { //:skip

    struct buf* bp;

    bp = bread(dev, 1);
    memmove(sb, bp->data, sizeof(*sb));
    brelse(bp);

    goto next(...);
}

__code iinitfs_impl(struct fs_impl* fs, __code next(...)) {

     initlock(&icache.lock, "icache");

    goto next(...);
}

__code iallocfs_impl(struct fs_impl* fs, uint dev, short type, __code next(...)) {
    goto allocinode(fs, dev, sb, next(...));
}

__code iupdatefs_impl(struct fs_impl* fs, struct inode* ip, __code next(...)) {
```

```

    struct buf *bp;
    struct dinode *dip;

    bp = bread(ip->dev, IBLOCK(ip->inum));

    dip = (struct dinode*) bp->data + ip->inum % IPB;
    dip->type = ip->type;
    dip->major = ip->major;
    dip->minor = ip->minor;
    dip->nlink = ip->nlink;
    dip->size = ip->size;

    memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
    log_write(bp);
    brelse(bp);

    goto next(...);
}

__code idupfs_impl(struct fs_impl* fs, struct inode* ip, __code next(...)) {

    acquire(&icache.lock);
    ip->ref++;
    release(&icache.lock);

    goto next(ip, ...);
}

__code ilockfs_impl(struct fs_impl* fs, struct inode* ip, __code next(...)) {

    goto lockinode1(fs, ip, bp, dip, next(...));
}

__code iunlockfs_impl(struct fs_impl* fs, struct inode* ip, __code next(...)) {

    if (ip == 0 || !(ip->flags & I_BUSY) || ip->ref < 1) {
       char* msg = "iunlock";
       struct Err* err = createKernelError(&proc->cbc_context);
       Gearef(cbc_context, Err)->msg = msg;
       goto meta(cbc_context, err->panic);
    }

    acquire(&icache.lock);
    ip->flags &= ~I_BUSY;
    wakeup(ip);
    release(&icache.lock);

    goto next(...);
}

__code iputfs_impl(struct fs_impl* fs, struct inode* ip, __code next(...)) {
    if (next == C_iputfs_impl) {
        next = fs->next2;
    }
    goto iput_check(fs, ip, next(...));
}

__code iunlockputfs_impl(struct fs_impl* fs, struct inode* ip, __code next(...)) {
    fs->next2 = next;
    goto iunlockfs_impl(ip, fs->iput, ...);
}

typedef struct stat stat;
__code statifs_impl(struct fs_impl* fs , struct inode* ip, struct stat* st, __code
    next(...)) { //:skip
```

```
    st->dev = ip->dev;
    st->ino = ip->inum;
    st->type = ip->type;
    st->nlink = ip->nlink;
    st->size = ip->size;
    goto next(...);
}

__code readifs_impl(struct fs_impl* fs, struct inode* ip, char* dst, uint off, uint tot, uint n, __code next(int ret, ...)) {
    if (ip->type == T_DEV) {
        goto readi_check_diskinode(fs, ip, dst, n, next(...));
    }

    if (off > ip->size || off + n < off) {
        ret = -1;
        goto next(ret, ...);
    }

    if (off + n > ip->size) {
        n = ip->size - off;
    }
    Gearef(cbc_context, fs)->tot = 0;
    goto readi_loopcheck(fs, tot, m, dst, off, n, next(...));
}

__code writeifs_impl(struct fs_impl* fs, struct inode* ip, char* src, uint off, uint tot, uint n, __code next(int ret, ...)) {
    if (ip->type == T_DEV) {
        goto writei_check_diskinode(fs, ip, src, n, next(...));
    }

    if (off > ip->size || off + n < off) {
        ret = -1;
        goto next(ret, ...);
    }

    if (off + n > MAXFILE * BSIZE) {
        ret = -1;
        goto next(ret, ...);
    }
    Gearef(cbc_context, fs)->tot = 0;
    goto writei_loopcheck(fs, tot, m, src, off, n, next(...));
}

__code namecmpfs_impl(struct fs_impl* fs, const char* s, const char* t, __code next(int strncmp_val, ...)) {
    strncmp_val = strncmp(s, t, DIRSIZ);
    goto next(strncmp_val, ...);
}

__code dirlookupfs_impl(struct fs_impl* fs, struct inode* dp, char* name, uint off, uint* poff, dirent* de, __code next(...)) { //:skip
    if (dp->type != T_DIR) {
       char* msg = "dirlookup␣not␣DIR";
       struct Err* err = createKernelError(&proc->cbc_context);
       Gearef(cbc_context, Err)->msg = msg;
       goto meta(cbc_context, err->panic);
    }
    Gearef(cbc_context, fs)->off = 0;
    goto dirlookup_loopcheck(fs, dp, name, off, poff, de, next(...));
}

__code dirlinkfs_impl(struct fs_impl* fs, struct inode* ip, struct dirent* de, struct inode* dp, char* name, uint off, uint inum, __code next(...)) { //:skip

```

```
220     if ((ip = dirlookup(dp, name, 0)) != 0) {
221         goto dirlink_namecheck(fs, ip, next(...));
222     }
223     Gearef(cbc_context, fs)->off = 0;
224     goto dirlink_loopcheck(fs, de, dp, off, next(...));
225 }
226 
227 __code nameifs_impl(struct fs_impl* fs, char* path, __code next(int namex_val, ...)) {
228     char name[DIRSIZ];
229     namex_val = namex(path, 0, name);
230     goto next(namex_val, ...);
231 }
232 
233 __code nameiparentfs_impl(struct fs_impl* fs, char* path, char* name, __code next(int namex_val, ...)) {
234 
235     namex_val = namex(path, 1, name);
236     goto next(namex_val, ...);
237 
238 }
```

2 行目のようにインターフェースのヘッダーファイルは #include ではなく #interface で呼び出す。4 行目の create fs_impl は Java などのコンストラクタに相当する。7 行目から 66 行目で Interface と実装の紐付けしている。CbC は 1 つ 1 つの Code Gear の信頼性を保障させるために細かくするべきであるため、 for 文や if 文がある場合はさらに実装を分ける。fs と同じように fs_impl を定義し、遷移 する関数名に対応させていく。分けた実装はさらに別で実装する (fs_impl_private.cbc)。

5.4 FileSystem の Interface の private な実装

インターフェースで定義した Code Gear 以外の Code Gaer も記述することができる。この Code Gear は基本的にインターフェースで指定された Code Gear 内からのみ継続さ れるため、Java の private メソッドのように扱われる。インターフェースと同じようにヘッダーファイルをソースコード 5.3 で定義する。

ソースコード 5.3: fs private のヘッダーファイル (fs_impl.h)

```
typedef struct fs_impl<Type, Isa> impl fs{
    __code allocinode(Type* fs_impl, uint dev, struct superblock* sb, __code next(...));
    __code allocinode_loop(Type* fs_impl, uint inum, uint dev, short type, struct superblock* sb, struct buf* bp, struct dinode* dip, __code next(...));
    __code allocinode_loopcheck(Type* fs_impl, uint inum, uint dev, struct superblock* sb, struct buf* bp, struct dinode* dip, __code next(...));
    __code allocinode_noloop(Type* fs_impl, uint inum, uint dev, short type, struct superblock* sb, struct buf* bp, struct dinode* dip, __code next(int ret, ...));
    __code lockinode1(Type* fs_impl, struct inode *ip, struct buf *bp, struct dinode *dip, __code next(...));
    __code lockinode2(Type* fs_impl, struct inode* ip, struct buf* bp, struct dinode* dip, __code next(...));
    __code lockinode_sleepcheck(Type* fs_impl, struct inode* ip, __code next(...));
    __code iput_check(Type* fs_impl, struct inode* ip, __code next(...));
    __code iput_inode_nolink(Type* fs_impl, struct inode* ip, __code next(...));
```

```
    __code readi_check_diskinode(Type* fs_impl,struct inode* ip, char* dst, uint n,
        next(int ret, ...));
    __code readi_loopcheck(Type* fs_impl, uint tot, uint m, char* dst, uint off, uint
         n, __code next(...));
    __code readi_loop(Type* fs_impl, struct inode *ip, struct buf* bp, uint tot, uint
         m, char* dst, uint off, uint n, __code next(...));
    __code readi_noloop(Type* fs_impl, uint n, __code next(int ret, ...));
    __code writei_check_diskinode(Type* fs_impl,struct inode* ip, char* src, uint n,
        __code next(int ret, ...));
    __code writei_loopcheck(Type* fs_impl, uint tot, uint m, char* src, uint off,
        uint n, __code next(...));
    __code writei_loop(Type* fs_impl, struct inode* ip, struct buf* bp, uint tot,
        uint m, char* src, uint off, uint n, __code next(...));
    __code writei_noloop(Type* fs_impl, struct inode* ip, uint n, uint off, __code
        next(int ret, ...));
    __code dirlookup_loopcheck(Type* fs_impl, struct inode* dp, char* name, uint off,
         uint* poff, dirent* de, next(...));
    __code dirlookup_loop(Type* fs_impl, struct inode* dp, char* name, uint off, uint
         inum, uint* poff, dirent* de, __code next(int ret, ...));
    __code dirlookup_noloop(Type* fs_impl, __code next(int ret, ...));
    __code dirlink_namecheck(Type* fs_impl, struct inode* ip, __code next(int ret,
        ...));
    __code dirlink_loopcheck(Type* fs_impl, struct dirent* de, struct inode* dp, uint
         off, __code next(...));
    __code dirlink_loop(Type* fs_impl, struct dirent* de, struct inode* dp, uint off,
         uint inum, __code next(...));
    __code dirlink_noloop(Type* fs_impl, struct dirent* de, struct inode* dp, uint
        off, uint inum, char* name, __code next(int ret, ...));
    __code next(...);
    __code next2(...);
} fs_impl;
```

第6章　まとめと今後の課題

今回の研究では xv6 の FileSystem 部分について CbC を用いて書き換えを行った。しかし、 xv6 は Gears OS を開発する前段階として開発しているので今後は書き換えた xv6 を Gears OS に適応した形に改良していく必要がある。xv6 の FileSystem 部分書き換え後 make し build することはできたが、デバックをまだ行っていないため正常に動くかどうか確認することが求められる。また、動かなかった場合修正を行い OS として機能しているか再確認する必要がある。後々は定理証明支援機 agda で証明できる OS として開発したい。

参考文献

[1] Kaito TOKKMORI and Shinji KONO. Implementing continuation based language in llvm and clang. *LOLA 2015*, July 2015.

[2] GNU Compiler Collection (GCC) Internals. *http://gcc.gnu.org/onlinedocs/gccint/*.

[3] 大城信康, 河野真治. Continuation based c の gcc4.6 上の実装について. 第 53 回プログラミング・シンポジウム予稿集, 第 2012 巻, pp. 69–78, jan 2012.

[4] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[5] Tokumori Kaito and Kono Shinji. Implementing continuation based language in llvm and clang. *LOLA 2015, Kyoto*, July 2015.

[6] 河野真治, 伊波立樹, 東恩納琢偉. Code gear、data gear に基づく os のプロトタイプ. 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 2016.

[7] Russ Cox, M Frans Kaashoek, and Robert Morris. Xv6, a simple unix-like teaching operating system, 2011. (2020 年 2 月 7 日閲覧).

[8] ARM Architecture Reference Manual. *http://infocenter.arm.com/help/topic/com.arm.doc.subset.architecture.reference/index.html*.

[9] 伊波立樹, 河野真治. Gears os の並列処理. 琉球大学工学部情報工学科平成 30 年度学位論文 (修士), 2018.

[10] 宮城光希, 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. 琉球大学工学部情報工学科平成 31 年度学位論文 (修士), 2019.

謝辞

本研究の遂行，また本論文の作成にあたり、御多忙にも関わらず終始懇切なる御指導と御教授を賜わりました河野真治准教授に深く感謝したします。

数々の貴重な御助言と細かな御配慮を戴いた並列信頼研究室の hoge 氏に深く感謝致します。

また一年間共に研究を行い、暖かな気遣いと励ましをもって支えてくれた並列信頼研究室の hoge 君、hoge 君、hoge さんに感謝致します。

最後に、有意義な時間を共に過ごした情報工学科の学友、並びに物心両面で支えてくれた両親に深く感謝致します。

2020 年 2 月

坂本昂弘