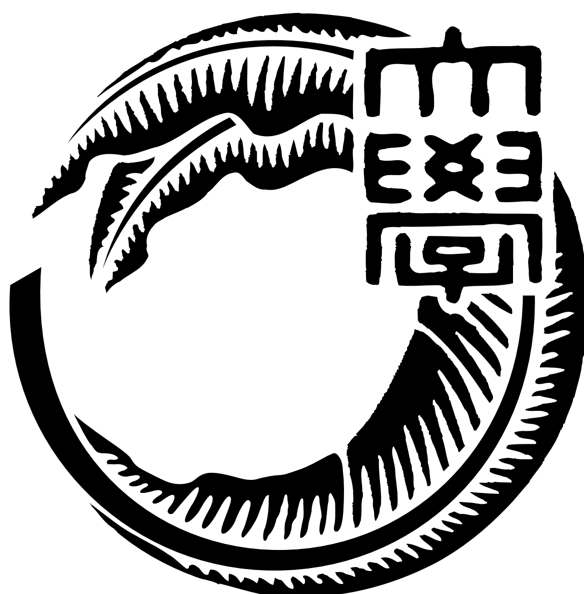
令和元年度 卒業論文

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

第1章　xv6 の OS の信頼性保証 **1**

第2章　Continuation based C **2**

2.1 Continuation based C の概要 . . . 2
2.2 CodeGear . . . 2
2.3 DataGear . . . 4
2.4 Meta CodeGear と Meta DataGear . . . 4

第3章　GearsOS **5**

3.1 GearsOS の概要 . . . 5
3.2 Context . . . 5
3.3 Inetrface . . . 6

第4章　xv6 **7**

4.1 xv6 の概要 . . . 7
4.2 xv6 の FileSystem . . . 7
4.3 FilleSystem の API . . . 8

第5章　CbC による FileSystem の書き換え **10**

5.1 書き換え方針 . . . 10
5.2 FileSystem の Interface の定義 . . . 10
5.3 FileSystem の Interface の実装 . . . 11
5.4 FileSystem の Interface の private な実装 . . . 14

第6章　まとめと今後の課題 **17**

図 目 次

2.1 CodeGear 間の継続 2
2.2 ソースコード 2.1 が表している CodeGear の状態遷移 3
2.3 CodeGear と DataGear 4
2.4 メタレベルの処理を可視化した CodeGear と DataGear 4

3.1 CodeGear、DataGear、contxt の関係図 5
3.2 Stack の Interface とその実装 6

4.1 xv6 の FileSystem Layer 8

5.1 xv6 FileSystem の Interface と実装 10
5.2 allocinode の ループによる遷移図 16

ソースコード目次

2.1 CodeGear の継続の例 3
5.1 FileSystem の Interface (fs.dg 一部抜粋) 10
5.2 FileSystem の Interface の実装 (fs_impl.cbc 一部抜粋) 11
5.3 fs private のヘッダーファイル (fs_impl.h 一部抜粋) 14
5.4 iallocfs_impl の実装 14
5.5 iallocfs_impl の private 実装 14
6.1 FileSystem の Interface 20
6.2 fs private のヘッダーファイル 20
6.3 fs Interface の実装 22
6.4 fs Interface の private 実装 27

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。

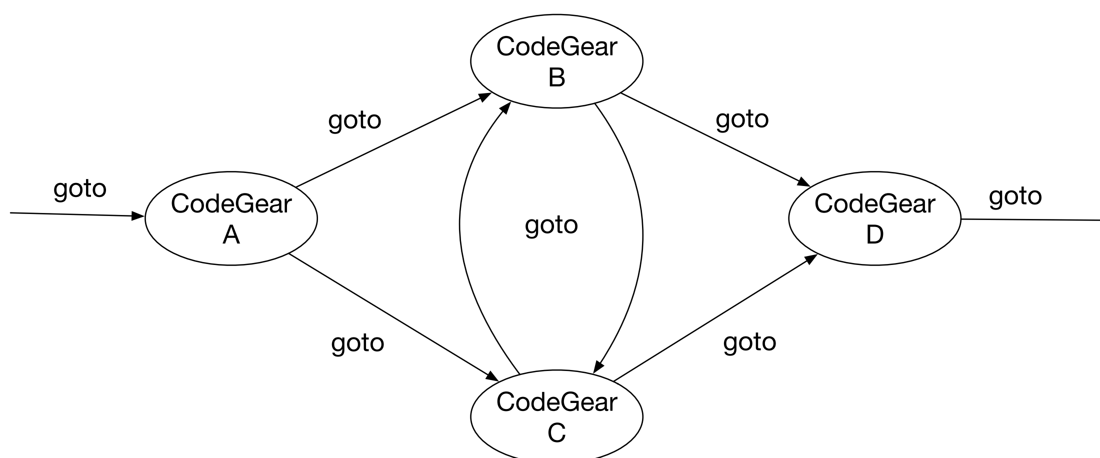


図 2.1: CodeGear 間の継続

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。

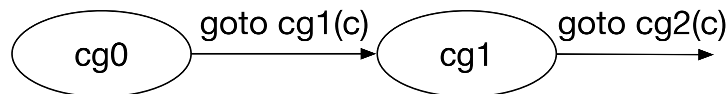


図 2.2: ソースコード 2.1 が表している CodeGear の状態遷移

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。

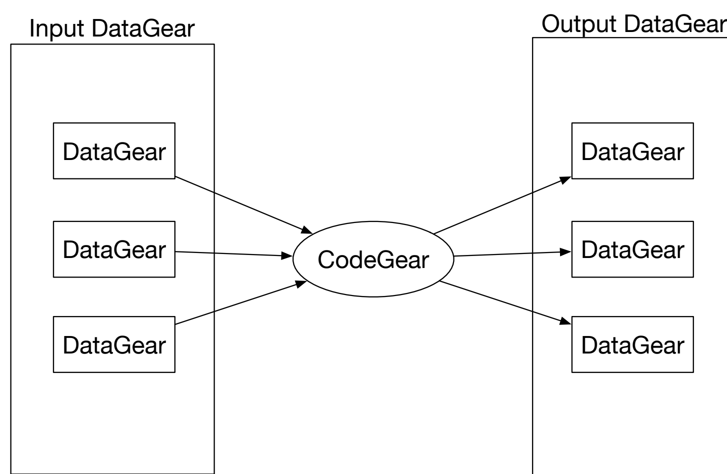


図 2.3: CodeGear と DataGear

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。

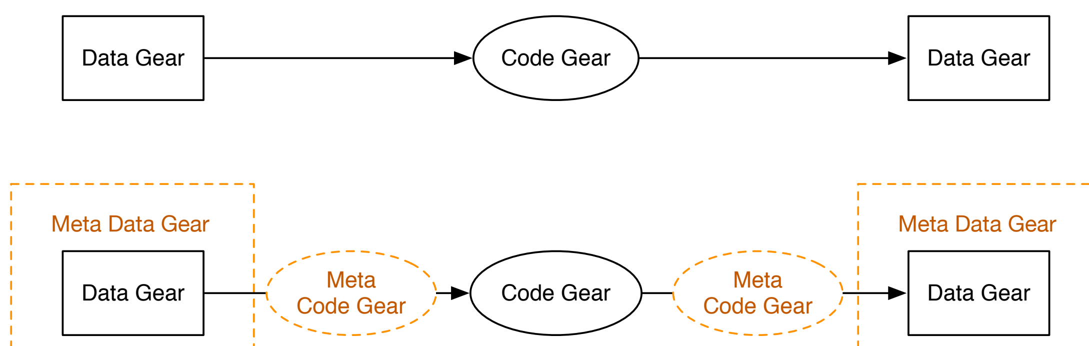


図 2.4: メタレベルの処理を可視化した CodeGear と DataGear

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。

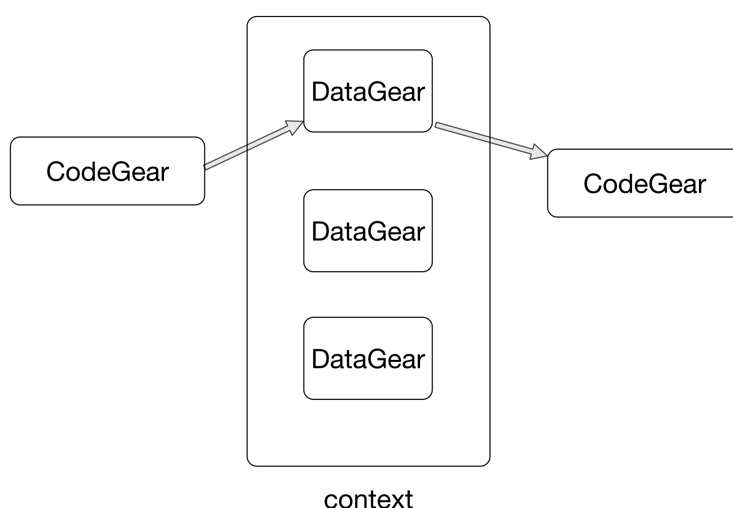


図 3.1: CodeGear、DataGear、contxt の関係図

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。

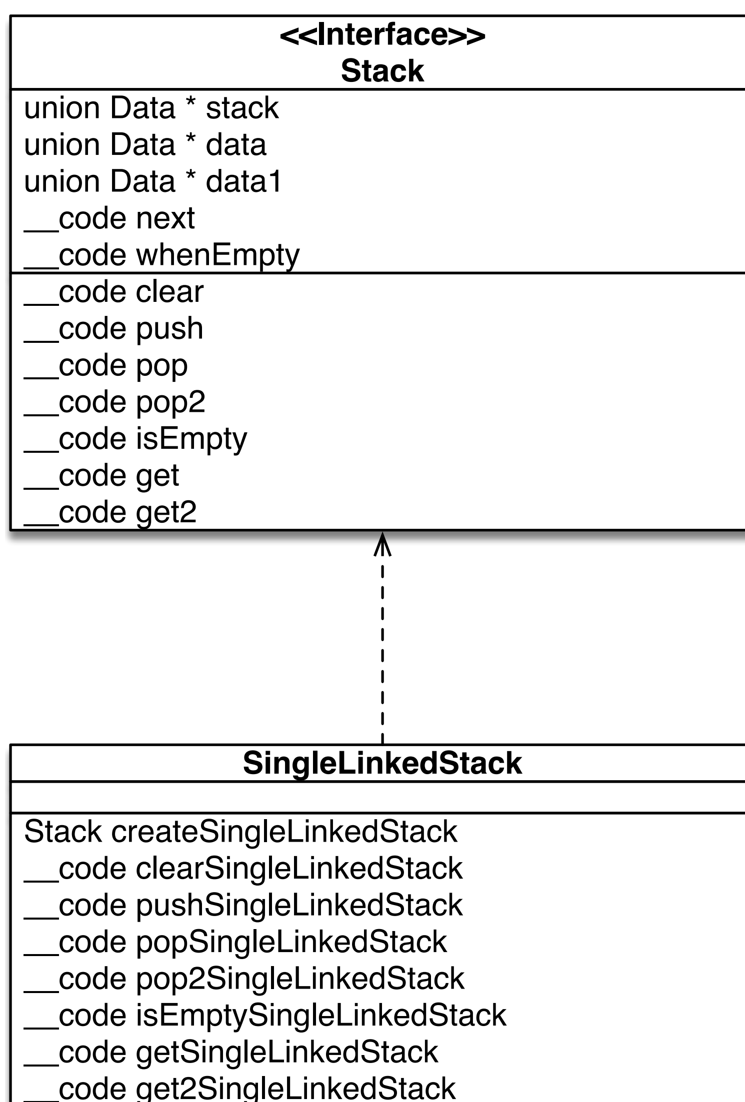


図 3.2: Stack の Interface とその実装

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。

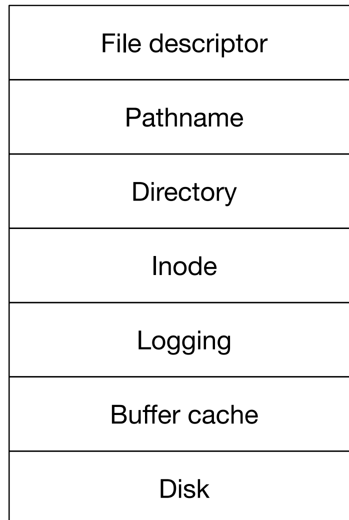


図 4.1: xv6 の FileSystem Layer

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

FileSystem の処理は複雑である。

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。

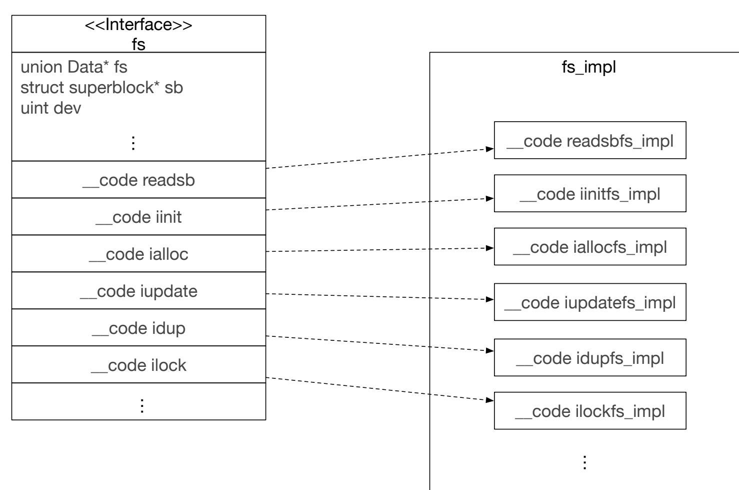


図 5.1: xv6 FileSystem の Interface と実装

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 一部抜粋)

```
1 typedef struct fs<Type,Impl> {
2     __code readsb(Impl* fs, uint dev, struct superblock* sb, __code next(...));
```

```
3      __code iinit(Impl* fs, __code next(...));
4      __code ialloc(Impl* fs, uint dev, short type, __code next(...));
5      __code iupdate(Impl* fs, struct inode* ip, __code next(...));
6      __code idup(Impl* fs, struct inode* ip, __code next(...));
7      __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
11 ....
12
13 } 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。CodeGear は __code CodeGear 名 (引数) の形で記述する。第一引数である Impl* fs が CodeGear の型、第二引数以降は Interface の実装時に利用する CodeGear が必要とする引数が入る。CodeGaer は 2 行目から 9 行目のように ”__code [CodeGear 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)

```
1  #interface "Err.h"
2  #interface "fs.dg"
3
4  fs* createfs_impl(struct Context* cbc_context) {
5      struct fs* fs = new fs();
6      struct fs_impl* fs_impl = new fs_impl();
7      fs->fs = (union Data*)fs_impl;
8      fs_impl->fs_impl = NULL;
9      fs_impl->sb = NULL;
10     fs_impl->ret = 0;
11     fs_impl->dev = 0;
12     fs_impl->type = 0;
13     fs_impl->bp = NULL;
14     fs_impl->dip = NULL;
15     fs_impl->inum = 0;
16     fs_impl->dp = NULL;
17     fs_impl->name = NULL;
18     fs_impl->off = 0;
19     fs_impl->poff = NULL;
20     fs_impl->de = NULL;
21     fs_impl->tot = 0;
22     fs_impl->m = 0;
23     fs_impl->dst = NULL;
24     fs_impl->n = 0;
25     fs_impl->src = NULL;
26     fs_impl->allocinode = C_allocinode;
27     fs_impl->allocinode_loop = C_allocinode_loop;
28     fs_impl->allocinode_loopcheck = C_allocinode_loopcheck;
29     fs_impl->allocinode_noloop = C_allocinode_noloop;
30     fs_impl->lockinode1 = C_lockinode1;
31     fs_impl->lockinode2 = C_lockinode2;
32     fs_impl->lockinode_sleepcheck = C_lockinode_sleepcheck;
33     fs_impl->iput_check = C_iput_check;
34     fs_impl->iput_inode_nolink = C_iput_inode_nolink;
35     fs_impl->readi_check_diskinode = C_readi_check_diskinode;
36     fs_impl->readi_loopcheck = C_readi_loopcheck;
```

```
37     fs_impl->readi_loop = C_readi_loop;
38     fs_impl->readi_noloop = C_readi_noloop;
39     fs_impl->writei_check_diskinode = C_writei_check_diskinode;
40     fs_impl->writei_loopcheck = C_writei_loopcheck;
41     fs_impl->writei_loop = C_writei_loop;
42     fs_impl->writei_noloop = C_writei_noloop;
43     fs_impl->dirlookup_loopcheck = C_dirlookup_loopcheck;
44     fs_impl->dirlookup_loop = C_dirlookup_loop;
45     fs_impl->dirlookup_noloop = C_dirlookup_noloop;
46     fs_impl->dirlink_namecheck = C_dirlink_namecheck;
47     fs_impl->dirlink_loopcheck = C_dirlink_loopcheck;
48     fs_impl->dirlink_loop = C_dirlink_loop;
49     fs_impl->dirlink_noloop = C_dirlink_noloop;
50     fs->readsb = C_readsbfs_impl;
51     fs->iinit = C_iinitfs_impl;
52     fs->ialloc = C_iallocfs_impl;
53     fs->iupdate = C_iupdatefs_impl;
54     fs->idup = C_idupfs_impl;
55     fs->ilock = C_ilockfs_impl;
56     fs->iunlock = C_iunlockfs_impl;
57     fs->iput = C_iputfs_impl;
58     fs->iunlockput = C_iunlockputfs_impl;
59     fs->stati = C_statifs_impl;
60     fs->readi = C_readifs_impl;
61     fs->writei = C_writeifs_impl;
62     fs->namecmp = C_namecmpfs_impl;
63     fs->dirlookup = C_dirlookupfs_impl;
64     fs->dirlink = C_dirlinkfs_impl;
65     fs->namei = C_nameifs_impl;
66     fs->nameiparent = C_nameiparentfs_impl;
67     return fs;
68 }
69
70 typedef struct superblock superblock;
71 __code readsbfs_impl(struct fs_impl* fs, uint dev, struct superblock* sb, __code
       next(...)) { //:skip
72
73     struct buf* bp;
74
75     bp = bread(dev, 1);
76     memmove(sb, bp->data, sizeof(*sb));
77     brelse(bp);
78
79     goto next(...);
80 }
81
82 __code iinitfs_impl(struct fs_impl* fs, __code next(...)) {
83
84      initlock(&icache.lock, "icache");
85
86     goto next(...);
87 }
88
89 __code iallocfs_impl(struct fs_impl* fs, uint dev, short type, __code next(...)) {
90     goto allocinode(fs, dev, sb, next(...));
91 }
92
93 __code iupdatefs_impl(struct fs_impl* fs, struct inode* ip, __code next(...)) {
94
95     struct buf *bp;
96     struct dinode *dip;
97
98     bp = bread(ip->dev, IBLOCK(ip->inum));
99
100    dip = (struct dinode*) bp->data + ip->inum % IPB;
101    dip->type = ip->type;
```

```
102     dip->major = ip->major;
103     dip->minor = ip->minor;
104     dip->nlink = ip->nlink;
105     dip->size = ip->size;
106
107     memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
108     log_write(bp);
109     brelse(bp);
110
111     goto next(...);
112 }
113
114 __code idupfs_impl(struct fs_impl* fs, struct inode* ip, __code next(...)) {
115
116     acquire(&icache.lock);
117     ip->ref++;
118     release(&icache.lock);
119
120     goto next(ip, ...);
121 }
122
123 __code ilockfs_impl(struct fs_impl* fs, struct inode* ip, __code next(...)) {
124
125     goto lockinode1(fs, ip, bp, dip, next(...));
126 }
127
128 __code iunlockfs_impl(struct fs_impl* fs, struct inode* ip, __code next(...)) {
129
130     if (ip == 0 || !(ip->flags & I_BUSY) || ip->ref < 1) {
131        char* msg = "iunlock";
132        struct Err* err = createKernelError(&proc->cbc_context);
133        Gearef(cbc_context, Err)->msg = msg;
134        goto meta(cbc_context, err->panic);
135     }
136
137     acquire(&icache.lock);
138     ip->flags &= ~I_BUSY;
139     wakeup(ip);
140     release(&icache.lock);
141
142     goto next(...);
143 }
144
145 __code iputfs_impl(struct fs_impl* fs, struct inode* ip, __code next(...)) {
146     if (next == C_iputfs_impl) {
147         next = fs->next2;
148     }
149     goto iput_check(fs, ip, next(...));
150 }
151
152 ....
```

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 CodeGear 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 CodeGear 以外の CodeGaer 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 一部抜粋)

```
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, ...)
        );

....

} fs_impl;
```

private での CbC の記述について説明する。記述量が多いため、if 文と for 文を書き換えた ialloc_impl という関数で説明する。書き換えコードは以下のソースコード??とソースコード 5.5 に示す。

ソースコード 5.4: iallocfs_impl の実装

```
#interface "fs.dg"

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

fs_impl.cbc の中で CodeGear である iallocfs_impl が処理されると goto で private な実装である fs_impl_private.cbc 側の allocinode という CodeGear へと遷移する。

ソースコード 5.5: iallocfs_impl の private 実装

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "stat.h"
#include "mmu.h"
#include "proc.h"
#include "spinlock.h"
#include "buf.h"
#include "fs.h"
#include "file.h"
#interface "fs_impl.h"
#interface "Err.h"
#define min(a, b) ((a) < (b) ? (a) : (b))

/*
fs_impl* createfs_impl2();
*/
```

```
18
19 __code allocinode(struct fs_impl* fs_impl, uint dev, struct superblock* sb, __code
       next(...)){ //:skip
20
21     readsb(dev, sb);
22     Gearef(cbc_context, fs_impl)->inum = 1;
23     goto allocinode_loopcheck(fs_impl, inum, dev, sb, bp, dip, next(...));
24
25 }
26
27 typedef struct buf buf;
28 typedef struct dinode dinode;
29
30 __code allocinode_loopcheck(struct fs_impl* fs_impl, uint inum, uint dev, struct
       superblock* sb, struct buf* bp, struct dinode* dip, __code next(...)){ //:skip
31     if( inum < sb->ninodes){
32         goto allocinode_loop(fs_impl, inum, dev, type, sb, bp, dip, next(...));
33     }
34     char* msg = "failed␣allocinode...";
35     struct Err* err = createKernelError(&proc->cbc_context);
36     Gearef(cbc_context, Err)->msg = msg;
37     goto meta(cbc_context, err->panic);
38
39 }
40
41 __code allocinode_loop(struct fs_impl* fs_impl, uint inum, uint dev, short type,
       struct superblock* sb, struct buf* bp, struct dinode* dip, __code next(...)){ //:
       skip
42     bp = bread(dev, IBLOCK(inum));
43     dip = (struct dinode*) bp->data + inum % IPB;
44     if(dip->type = 0){
45         goto allocinode_noloop(fs_impl, inum, dev, sb, bp, dip, next(...));
46     }
47
48     brelse(bp);
49     inum++;
50     goto allocinode_loopcheck(fs_impl, inum, dev, type, sb, bp, dip, next(...));
51 }
52
53 struct {
54     struct spinlock lock;
55     struct inode inode[NINODE];
56 } icache;
57
58 static struct inode* iget (uint dev, uint inum)
59 {
60     struct inode *ip, *empty;
61
62     acquire(&icache.lock);
63
64     empty = 0;
65
66     for (ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++) {
67         if (ip->ref > 0 && ip->dev == dev && ip->inum == inum) {
68             ip->ref++;
69             release(&icache.lock);
70             return ip;
71         }
72
73         if (empty == 0 && ip->ref == 0) {
74             empty = ip;
75         }
76     }
77
78     if (empty == 0) {
79         panic("iget:␣no␣inodes");
```

```
80      }
81
82      ip = empty;
83      ip->dev = dev;
84      ip->inum = inum;
85      ip->ref = 1;
86      ip->flags = 0;
87      release(&icache.lock);
88
89      return ip;
90  }
91
92  __code allocinode_noloop(struct fs_impl* fs_impl, uint inum, uint dev, short type,
        struct superblock* sb, struct buf* bp, struct dinode* dip, __code next(int ret,
        ...)){ //:skip
93
94      memset(dip, 0, sizeof(*dip));
95      dip->type = type;
96      log_write(bp);
97      brelse(bp);
98
99      ret = iget(dev, inum);
100     goto next(ret, ...);
101
102 }
```

private 側では iallocfs_impl からの受け皿としての allocinode、for 文のループ条件を確認する allocinode_loopcheck、ループした際の処理をする allocinode_loop、ループから抜けた際の処理をする allocinode_noloop の 4 つの CodeGear から成り立っている。 ループの条件に当てはまらない際は panic へと処理が移り処理が停止する。これらの CodeGear の一連の挙動を図 5.2 に示す。

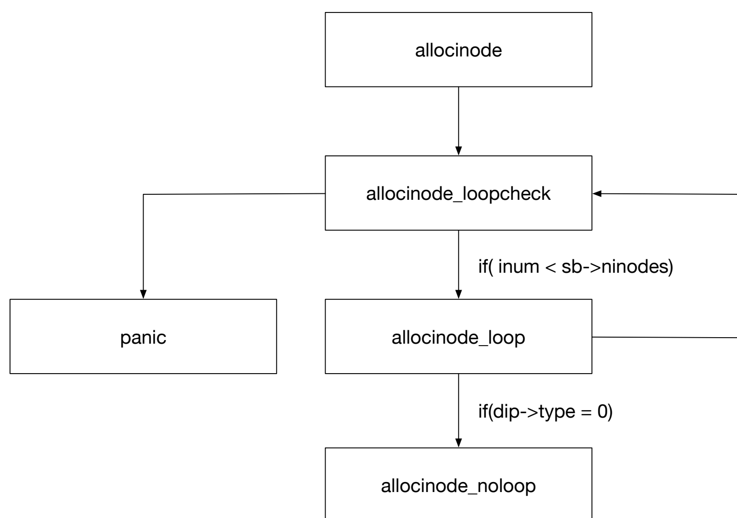


図 5.2: allocinode の ループによる遷移図

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

付録

ソースコード 6.1: FileSystem の Interface

```
typedef struct fs<Type,Impl> {
    union Data* fs;
    struct superblock* sb;
    uint dev;
    short type;
    struct inode* ip;
    struct stat* st;
    char* dst;
    uint off;
    uint n;
    const char* s;
    const char* t;
    struct inode* dp;
    char* name;
    uint* poff;
    uint inum;
    char* path;
    char* src;
    int namex_val;
    int strncmp_val;
    dirent* de;
    int ret;
    uint tot;
    __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(...));
    __code iunlock(Impl* fs, struct inode* ip, __code next(...));
    __code iput(Impl* fs, struct inode* ip, __code next(...));
    __code iunlockput(Impl* fs, struct inode* ip, __code next(...));
    __code stati(Impl* fs , struct inode* ip, struct stat* st, __code next(...));
    __code readi(Impl* fs, struct inode* ip, char* dst, uint off, uint tot, uint n,
        __code next(int ret, ...));
    __code writei(Impl* fs, struct inode* ip, char* src, uint off, uint tot, uint n,
        __code next(int ret, ...));
    __code namecmp(Impl* fs, const char* s, const char* t, __code next(int
        strncmp_val, ...));
    __code dirlookup(Impl* fs, struct inode* dp, char* name, uint off, uint* poff,
        dirent* de, __code next(int ret, ...));
    __code dirlink(struct fs_impl* fs, struct inode* ip, struct dirent* de, struct
        inode* dp, char* name, uint off, uint inum, __code next(...));
    __code namei(Impl* fs, char* path, __code next(int namex_val, ...));
    __code nameiparent(Impl* fs, char* path, char* name, __code next(int namex_val,
        ...));
    __code next(...);
} fs;
```

ソースコード 6.2: fs private のヘッダーファイル

```
1 typedef struct fs_impl<Type, Isa> impl fs{
2     union Data* fs_impl;
3     struct superblock* sb;
4     int ret;
5     uint dev;
6     short type;
7     struct buf* bp;
8     struct dinode* dip;
9     uint inum;
10    struct inode* dp;
11    char* name;
12    uint off;
13    uint* poff;
14    dirent* de;
15    uint tot;
16    uint m;
17    char* dst;
18    uint n;
19    char* src;
20
21    __code allocinode(Type* fs_impl, uint dev, struct superblock* sb, __code next
          (...));
22    __code allocinode_loop(Type* fs_impl, uint inum, uint dev, short type, struct
          superblock* sb, struct buf* bp, struct dinode* dip, __code next(...));
23    __code allocinode_loopcheck(Type* fs_impl, uint inum, uint dev, struct superblock
          * sb, struct buf* bp, struct dinode* dip, __code next(...));
24    __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, ...)
          );
25    __code lockinode1(Type* fs_impl, struct inode *ip, struct buf *bp, struct dinode
          *dip, __code next(...));
26    __code lockinode2(Type* fs_impl, struct inode* ip, struct buf* bp, struct dinode*
           dip, __code next(...));
27    __code lockinode_sleepcheck(Type* fs_impl, struct inode* ip, __code next(...));
28    __code iput_check(Type* fs_impl, struct inode* ip, __code next(...));
29    __code iput_inode_nolink(Type* fs_impl, struct inode* ip, __code next(...));
30    __code readi_check_diskinode(Type* fs_impl,struct inode* ip, char* dst, uint n,
          next(int ret, ...));
31    __code readi_loopcheck(Type* fs_impl, uint tot, uint m, char* dst, uint off, uint
           n, __code next(...));
32    __code readi_loop(Type* fs_impl, struct inode *ip, struct buf* bp, uint tot, uint
           m, char* dst, uint off, uint n, __code next(...));
33    __code readi_noloop(Type* fs_impl, uint n, __code next(int ret, ...));
34    __code writei_check_diskinode(Type* fs_impl,struct inode* ip, char* src, uint n,
          __code next(int ret, ...));
35    __code writei_loopcheck(Type* fs_impl, uint tot, uint m, char* src, uint off,
          uint n, __code next(...));
36    __code writei_loop(Type* fs_impl, struct inode* ip, struct buf* bp, uint tot,
          uint m, char* src, uint off, uint n, __code next(...));
37    __code writei_noloop(Type* fs_impl, struct inode* ip, uint n, uint off, __code
          next(int ret, ...));
38    __code dirlookup_loopcheck(Type* fs_impl, struct inode* dp, char* name, uint off,
           uint* poff, dirent* de, next(...));
39    __code dirlookup_loop(Type* fs_impl, struct inode* dp, char* name, uint off, uint
           inum, uint* poff, dirent* de, __code next(int ret, ...));
40    __code dirlookup_noloop(Type* fs_impl, __code next(int ret, ...));
41    __code dirlink_namecheck(Type* fs_impl, struct inode* ip, __code next(int ret,
          ...));
42    __code dirlink_loopcheck(Type* fs_impl, struct dirent* de, struct inode* dp, uint
           off, __code next(...));
43    __code dirlink_loop(Type* fs_impl, struct dirent* de, struct inode* dp, uint off,
           uint inum, __code next(...));
44    __code dirlink_noloop(Type* fs_impl, struct dirent* de, struct inode* dp, uint
          off, uint inum, char* name, __code next(int ret, ...));
```

```
45    __code next(...);
46    __code next2(...);
47 } fs_impl;
```

ソースコード 6.3: fs Interface の実装

```
1 #include "types.h"
2 #include "defs.h"
3 #include "param.h"
4 #include "stat.h"
5 #include "mmu.h"
6 #include "proc.h"
7 #include "spinlock.h"
8 #include "buf.h"
9 #include "fs.h"
10 #include "file.h"
11 #interface "Err.h"
12 #interface "fs.dg"
13
14 fs* createfs_impl(struct Context* cbc_context) {
15     struct fs* fs = new fs();
16     struct fs_impl* fs_impl = new fs_impl();
17     fs->fs = (union Data*)fs_impl;
18     fs_impl->fs_impl = NULL;
19     fs_impl->sb = NULL;
20     fs_impl->ret = 0;
21     fs_impl->dev = 0;
22     fs_impl->type = 0;
23     fs_impl->bp = NULL;
24     fs_impl->dip = NULL;
25     fs_impl->inum = 0;
26     fs_impl->dp = NULL;
27     fs_impl->name = NULL;
28     fs_impl->off = 0;
29     fs_impl->poff = NULL;
30     fs_impl->de = NULL;
31     fs_impl->tot = 0;
32     fs_impl->m = 0;
33     fs_impl->dst = NULL;
34     fs_impl->n = 0;
35     fs_impl->src = NULL;
36     fs_impl->allocinode = C_allocinode;
37     fs_impl->allocinode_loop = C_allocinode_loop;
38     fs_impl->allocinode_loopcheck = C_allocinode_loopcheck;
39     fs_impl->allocinode_noloop = C_allocinode_noloop;
40     fs_impl->lockinode1 = C_lockinode1;
41     fs_impl->lockinode2 = C_lockinode2;
42     fs_impl->lockinode_sleepcheck = C_lockinode_sleepcheck;
43     fs_impl->iput_check = C_iput_check;
44     fs_impl->iput_inode_nolink = C_iput_inode_nolink;
45     fs_impl->readi_check_diskinode = C_readi_check_diskinode;
46     fs_impl->readi_loopcheck = C_readi_loopcheck;
47     fs_impl->readi_loop = C_readi_loop;
48     fs_impl->readi_noloop = C_readi_noloop;
49     fs_impl->writei_check_diskinode = C_writei_check_diskinode;
50     fs_impl->writei_loopcheck = C_writei_loopcheck;
51     fs_impl->writei_loop = C_writei_loop;
52     fs_impl->writei_noloop = C_writei_noloop;
53     fs_impl->dirlookup_loopcheck = C_dirlookup_loopcheck;
54     fs_impl->dirlookup_loop = C_dirlookup_loop;
55     fs_impl->dirlookup_noloop = C_dirlookup_noloop;
56     fs_impl->dirlink_namecheck = C_dirlink_namecheck;
57     fs_impl->dirlink_loopcheck = C_dirlink_loopcheck;
58     fs_impl->dirlink_loop = C_dirlink_loop;
59     fs_impl->dirlink_noloop = C_dirlink_noloop;
```

```
60     fs->readsb = C_readsbfs_impl;
61     fs->iinit = C_iinitfs_impl;
62     fs->ialloc = C_iallocfs_impl;
63     fs->iupdate = C_iupdatefs_impl;
64     fs->idup = C_idupfs_impl;
65     fs->ilock = C_ilockfs_impl;
66     fs->iunlock = C_iunlockfs_impl;
67     fs->iput = C_iputfs_impl;
68     fs->iunlockput = C_iunlockputfs_impl;
69     fs->stati = C_statifs_impl;
70     fs->readi = C_readifs_impl;
71     fs->writei = C_writeifs_impl;
72     fs->namecmp = C_namecmpfs_impl;
73     fs->dirlookup = C_dirlookupfs_impl;
74     fs->dirlink = C_dirlinkfs_impl;
75     fs->namei = C_nameifs_impl;
76     fs->nameiparent = C_nameiparentfs_impl;
77     return fs;
78 }
79
80 typedef struct superblock superblock;
81 __code readsbfs_impl(struct fs_impl* fs, uint dev, struct superblock* sb, __code
       next(...)) { //:skip
82
83     struct buf* bp;
84
85     bp = bread(dev, 1);
86     memmove(sb, bp->data, sizeof(*sb));
87     brelse(bp);
88
89     goto next(...);
90 }
91
92 struct {
93     struct spinlock lock;
94     struct inode inode[NINODE];
95 } icache;
96
97 __code iinitfs_impl(struct fs_impl* fs, __code next(...)) {
98
99      initlock(&icache.lock, "icache");
100
101     goto next(...);
102 }
103
104 __code iallocfs_impl(struct fs_impl* fs, uint dev, short type, __code next(...)) {
105     goto allocinode(fs, dev, sb, next(...));
106 }
107
108 __code iupdatefs_impl(struct fs_impl* fs, struct inode* ip, __code next(...)) {
109
110     struct buf *bp;
111     struct dinode *dip;
112
113     bp = bread(ip->dev, IBLOCK(ip->inum));
114
115     dip = (struct dinode*) bp->data + ip->inum % IPB;
116     dip->type = ip->type;
117     dip->major = ip->major;
118     dip->minor = ip->minor;
119     dip->nlink = ip->nlink;
120     dip->size = ip->size;
121
122     memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
123     log_write(bp);
124     brelse(bp);
```

```
125
126     goto next(...);
127 }
128
129 __code idupfs_impl(struct fs_impl* fs, struct inode* ip, __code next(...)) {
130
131     acquire(&icache.lock);
132     ip->ref++;
133     release(&icache.lock);
134
135     goto next(ip, ...);
136
137 }
138
139 __code ilockfs_impl(struct fs_impl* fs, struct inode* ip, __code next(...)) {
140
141     goto lockinode1(fs, ip, bp, dip, next(...));
142 }
143
144 __code iunlockfs_impl(struct fs_impl* fs, struct inode* ip, __code next(...)) {
145
146     if (ip == 0 || !(ip->flags & I_BUSY) || ip->ref < 1) {
147        char* msg = "iunlock";
148        struct Err* err = createKernelError(&proc->cbc_context);
149        Gearef(cbc_context, Err)->msg = msg;
150        goto meta(cbc_context, err->panic);
151     }
152
153     acquire(&icache.lock);
154     ip->flags &= ~I_BUSY;
155     wakeup(ip);
156     release(&icache.lock);
157
158     goto next(...);
159 }
160
161 __code iputfs_impl(struct fs_impl* fs, struct inode* ip, __code next(...)) {
162     if (next == C_iputfs_impl) {
163         next = fs->next2;
164     }
165     goto iput_check(fs, ip, next(...));
166 }
167
168 __code iunlockputfs_impl(struct fs_impl* fs, struct inode* ip, __code next(...)) {
169     fs->next2 = next;
170     goto iunlockfs_impl(ip, fs->iput, ...);
171 }
172
173 typedef struct stat stat;
174 __code statifs_impl(struct fs_impl* fs , struct inode* ip, struct stat* st, __code
        next(...)) { //:skip
175     st->dev = ip->dev;
176     st->ino = ip->inum;
177     st->type = ip->type;
178     st->nlink = ip->nlink;
179     st->size = ip->size;
180     goto next(...);
181 }
182
183 __code readifs_impl(struct fs_impl* fs, struct inode* ip, char* dst, uint off, uint
        tot, uint n, __code next(int ret, ...)) {
184     if (ip->type == T_DEV) {
185         goto readi_check_diskinode(fs, ip, dst, n, next(...));
186     }
187
188     if (off > ip->size || off + n < off) {
```

```
189         ret = -1;
190         goto next(ret, ...);
191     }
192
193     if (off + n > ip->size) {
194         n = ip->size - off;
195     }
196     Gearef(cbc_context, fs)->tot = 0;
197     goto readi_loopcheck(fs, tot, m, dst, off, n, next(...));
198 }
199
200 __code writeifs_impl(struct fs_impl* fs, struct inode* ip, char* src, uint off, uint
         tot, uint n, __code next(int ret, ...)) {
201     if (ip->type == T_DEV) {
202         goto writei_check_diskinode(fs, ip, src, n, next(...));
203     }
204
205     if (off > ip->size || off + n < off) {
206         ret = -1;
207         goto next(ret, ...);
208     }
209
210     if (off + n > MAXFILE * BSIZE) {
211         ret = -1;
212         goto next(ret, ...);
213     }
214     Gearef(cbc_context, fs)->tot = 0;
215     goto writei_loopcheck(fs, tot, m, src, off, n, next(...));
216 }
217
218
219 __code namecmpfs_impl(struct fs_impl* fs, const char* s, const char* t, __code next(
        int strncmp_val, ...)) {
220     strncmp_val = strncmp(s, t, DIRSIZ);
221     goto next(strncmp_val, ...);
222 }
223
224 __code dirlookupfs_impl(struct fs_impl* fs, struct inode* dp, char* name, uint off,
        uint* poff, dirent* de, __code next(...)) { //:skip
225     if (dp->type != T_DIR) {
226        char* msg = "dirlookup not DIR";
227        struct Err* err = createKernelError(&proc->cbc_context);
228        Gearef(cbc_context, Err)->msg = msg;
229        goto meta(cbc_context, err->panic);
230     }
231     Gearef(cbc_context, fs)->off = 0;
232     goto dirlookup_loopcheck(fs, dp, name, off, poff, de, next(...));
233 }
234
235 __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
236     if ((ip = dirlookup(dp, name, 0)) != 0) {
237         goto dirlink_namecheck(fs, ip, next(...));
238     }
239     Gearef(cbc_context, fs)->off = 0;
240     goto dirlink_loopcheck(fs, de, dp, off, next(...));
241 }
242
243 static struct inode* iget (uint dev, uint inum)
244 {
245     struct inode *ip, *empty;
246
247     acquire(&icache.lock);
248
249     empty = 0;
250
```

```
251     for (ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++) {
252         if (ip->ref > 0 && ip->dev == dev && ip->inum == inum) {
253             ip->ref++;
254             release(&icache.lock);
255             return ip;
256         }
257 
258         if (empty == 0 && ip->ref == 0) { // Remember empty slot.
259             empty = ip;
260         }
261     }
262     if (empty == 0) {
263         panic("iget:␣no␣inodes");
264     }
265 
266     ip = empty;
267     ip->dev = dev;
268     ip->inum = inum;
269     ip->ref = 1;
270     ip->flags = 0;
271     release(&icache.lock);
272 
273     return ip;
274 }
275 
276 static char* skipelem (char *path, char *name)
277 {
278     char *s;
279     int len;
280 
281     while (*path == '/') {
282         path++;
283     }
284 
285     if (*path == 0) {
286         return 0;
287     }
288 
289     s = path;
290 
291     while (*path != '/' && *path != 0) {
292         path++;
293     }
294 
295     len = path - s;
296 
297     if (len >= DIRSIZ) {
298         memmove(name, s, DIRSIZ);
299     } else {
300         memmove(name, s, len);
301         name[len] = 0;
302     }
303 
304     while (*path == '/') {
305         path++;
306     }
307 
308     return path;
309 }
310 
311 
312 static struct inode* namex (char *path, int nameiparent, char *name)
313 {
314     struct inode *ip, *next;
315 
316     if (*path == '/') {
```

```
317         ip = iget(ROOTDEV, ROOTINO);
318     } else {
319         ip = idup(proc->cwd);
320     }
321 
322     while ((path = skipelem(path, name)) != 0) {
323         ilock(ip);
324 
325         if (ip->type != T_DIR) {
326             iunlockput(ip);
327             return 0;
328         }
329 
330         if (nameiparent && *path == '\0') {
331             iunlock(ip);
332             return ip;
333         }
334 
335         if ((next = dirlookup(ip, name, 0)) == 0) {
336             iunlockput(ip);
337             return 0;
338         }
339 
340         iunlockput(ip);
341         ip = next;
342     }
343 
344     if (nameiparent) {
345         iput(ip);
346         return 0;
347     }
348 
349     return ip;
350 }
351 
352 __code nameifs_impl(struct fs_impl* fs, char* path, __code next(int namex_val, ...))
      {
353     char name[DIRSIZ];
354     namex_val = namex(path, 0, name);
355     goto next(namex_val, ...);
356 }
357 
358 __code nameiparentfs_impl(struct fs_impl* fs, char* path, char* name, __code next(
    int namex_val, ...)) {
359 
360     namex_val = namex(path, 1, name);
361     goto next(namex_val, ...);
362 
363 }
```

ソースコード 6.4: fs Interface の private 実装

```
1 #include "types.h"
2 #include "defs.h"
3 #include "param.h"
4 #include "stat.h"
5 #include "mmu.h"
6 #include "proc.h"
7 #include "spinlock.h"
8 #include "buf.h"
9 #include "fs.h"
10 #include "file.h"
11 #interface "fs_impl.h"
12 #interface "Err.h"
13 #define min(a, b) ((a) < (b) ? (a) : (b))
```

```
14
15 /*
16 fs_impl* createfs_impl2();
17 */
18
19 __code allocinode(struct fs_impl* fs_impl, uint dev, struct superblock* sb, __code
       next(...)){ //:skip
20
21     readsb(dev, sb);
22     Gearef(cbc_context, fs_impl)->inum = 1;
23     goto allocinode_loopcheck(fs_impl, inum, dev, sb, bp, dip, next(...));
24
25 }
26
27 typedef struct buf buf;
28 typedef struct dinode dinode;
29
30 __code allocinode_loopcheck(struct fs_impl* fs_impl, uint inum, uint dev, struct
       superblock* sb, struct buf* bp, struct dinode* dip, __code next(...)){ //:skip
31     if( inum < sb->ninodes){
32         goto allocinode_loop(fs_impl, inum, dev, type, sb, bp, dip, next(...));
33     }
34     char* msg = "failed␣allocinode...";
35     struct Err* err = createKernelError(&proc->cbc_context);
36     Gearef(cbc_context, Err)->msg = msg;
37     goto meta(cbc_context, err->panic);
38
39 }
40
41 __code allocinode_loop(struct fs_impl* fs_impl, uint inum, uint dev, short type,
       struct superblock* sb, struct buf* bp, struct dinode* dip, __code next(...)){ //:
       skip
42     bp = bread(dev, IBLOCK(inum));
43     dip = (struct dinode*) bp->data + inum % IPB;
44     if(dip->type = 0){
45         goto allocinode_noloop(fs_impl, inum, dev, sb, bp, dip, next(...));
46     }
47
48     brelse(bp);
49     inum++;
50     goto allocinode_loopcheck(fs_impl, inum, dev, type, sb, bp, dip, next(...));
51 }
52
53 struct {
54     struct spinlock lock;
55     struct inode inode[NINODE];
56 } icache;
57
58 static struct inode* iget (uint dev, uint inum)
59 {
60     struct inode *ip, *empty;
61
62     acquire(&icache.lock);
63
64     empty = 0;
65
66     for (ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++) {
67         if (ip->ref > 0 && ip->dev == dev && ip->inum == inum) {
68             ip->ref++;
69             release(&icache.lock);
70             return ip;
71         }
72
73         if (empty == 0 && ip->ref == 0) { // Remember empty slot.
74             empty = ip;
75         }
```

```
76      }
77
78      if (empty == 0) {
79          panic("iget:␣no␣inodes");
80      }
81
82      ip = empty;
83      ip->dev = dev;
84      ip->inum = inum;
85      ip->ref = 1;
86      ip->flags = 0;
87      release(&icache.lock);
88
89      return ip;
90  }
91
92  __code allocinode_noloop(struct fs_impl* fs_impl, uint inum, uint dev, short type,
        struct superblock* sb, struct buf* bp, struct dinode* dip, __code next(int ret,
        ...)){ //:skip
93
94      memset(dip, 0, sizeof(*dip));
95      dip->type = type;
96      log_write(bp);
97      brelse(bp);
98
99      ret = iget(dev, inum);
100     goto next(ret, ...);
101
102 }
103
104 __code lockinode1(struct fs_impl* fs_impl, struct inode* ip, struct buf* bp, struct
        dinode* dip, __code next(...)){ //:skip
105
106     if (ip == 0 || ip->ref < 1) {
107        char* msg = "ilock";
108        struct Err* err = createKernelError(&proc->cbc_context);
109        Gearef(cbc_context, Err)->msg = msg;
110        goto meta(cbc_context, err->panic);
111      }
112     acquire(&icache.lock);
113
114     goto lockinode_sleepcheck(fs_impl, ip, next(...));
115
116 }
117
118
119 __code lockinode2(struct fs_impl* fs_impl, struct inode* ip, struct buf* bp, struct
        dinode* dip, __code next(...)){ //:skip
120
121     ip->flags |= I_BUSY;
122     release(&icache.lock);
123
124     if (!(ip->flags & I_VALID)) {
125         bp = bread(ip->dev, IBLOCK(ip->inum));
126
127         dip = (struct dinode*) bp->data + ip->inum % IPB;
128         ip->type = dip->type;
129         ip->major = dip->major;
130         ip->minor = dip->minor;
131         ip->nlink = dip->nlink;
132         ip->size = dip->size;
133
134         memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
135         brelse(bp);
136         ip->flags |= I_VALID;
137
```

```
138         if (ip->type == 0) {
139             char* msg = "ilock:␣no␣type";
140             struct Err* err = createKernelError(&proc->cbc_context);
141             Gearef(cbc_context, Err)->msg = msg;
142             goto meta(cbc_context, err->panic);
143         }
144     }
145     goto next(...);
146 }
147 __code lockinode_sleepcheck(struct fs_impl* fs_impl, struct inode* ip, __code next
        (...)){
148     if(ip->flags & I_BUSY){
149         sleep(ip, &icache.lock);
150         goto lockinode_sleepcheck(fs_impl, ip, next(...));
151     }
152     goto lockinode2(fs_impl, ip, bp, dip, next(...));
153 }
154
155 __code iput_check(struct fs_impl* fs_impl, struct inode* ip, __code next(...)){
156     acquire(&icache.lock);
157     if (ip->ref == 1 && (ip->flags & I_VALID) && ip->nlink == 0) {
158         goto iput_inode_nolink(fs_impl, ip, next(...));
159     }
160     ip->ref--;
161     release(&icache.lock);
162     goto next(...);
163
164 }
165
166 static void bfree (int dev, uint b)
167 {
168     struct buf *bp;
169     struct superblock sb;
170     int bi, m;
171
172     readsb(dev, &sb);
173     bp = bread(dev, BBLOCK(b, sb.ninodes));
174     bi = b % BPB;
175     m = 1 << (bi % 8);
176
177     if ((bp->data[bi / 8] & m) == 0) {
178         panic("freeing␣free␣block");
179     }
180
181     bp->data[bi / 8] &= ~m;
182     log_write(bp);
183     brelse(bp);
184 }
185
186
187 static void itrunc (struct inode *ip)
188 {
189     int i, j;
190     struct buf *bp;
191     uint *a;
192
193     for (i = 0; i < NDIRECT; i++) {
194         if (ip->addrs[i]) {
195             bfree(ip->dev, ip->addrs[i]);
196             ip->addrs[i] = 0;
197         }
198     }
199
200     if (ip->addrs[NDIRECT]) {
201         bp = bread(ip->dev, ip->addrs[NDIRECT]);
202         a = (uint*) bp->data;
```

```
203
204         for (j = 0; j < NINDIRECT; j++) {
205             if (a[j]) {
206                 bfree(ip->dev, a[j]);
207             }
208         }
209
210         brelse(bp);
211         bfree(ip->dev, ip->addrs[NDIRECT]);
212         ip->addrs[NDIRECT] = 0;
213     }
214
215     ip->size = 0;
216     iupdate(ip);
217 }
218
219 __code iput_inode_nolink(struct fs_impl* fs_impl, struct inode* ip, __code next(...)
        ){
220
221     if (ip->flags & I_BUSY) {
222         char* msg = "iput␣busy";
223         struct Err* err = createKernelError(&proc->cbc_context);
224         Gearef(cbc_context, Err)->msg = msg;
225         goto meta(cbc_context, err->panic);
226     }
227
228     ip->flags |= I_BUSY;
229     release(&icache.lock);
230     itrunc(ip);
231     ip->type = 0;
232     iupdate(ip);
233
234     acquire(&icache.lock);
235     ip->flags = 0;
236     wakeup(ip);
237     goto next(...);
238 }
239
240 __code readi_check_diskinode(struct fs_impl* fs_impl,struct inode* ip, char* dst,
        uint n, __code next(int ret, ...)){
241     if (ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read) {
242         ret = -1;
243         goto next(ret, ...);
244     }
245
246     ret = devsw[ip->major].read(ip, dst, n);
247     goto next(ret, ...);
248 }
249
250 __code readi_loopcheck(struct fs_impl* fs_impl, uint tot, uint m, char* dst, uint
        off, uint n, __code next(...)){
251     if(tot < n){
252         goto readi_loop(fs_impl, ip, bp, tot, m, dst, off, n, next(...));
253     }
254     goto readi_noloop(fs_impl, next(...));
255 }
256
257 static void bzero (int dev, int bno)
258 {
259     struct buf *bp;
260
261     bp = bread(dev, bno);
262     memset(bp->data, 0, BSIZE);
263     log_write(bp);
264     brelse(bp);
265 }
```

```
266
267 static uint balloc (uint dev)
268 {
269     int b, bi, m;
270     struct buf *bp;
271     struct superblock sb;
272
273     bp = 0;
274     readsb(dev, &sb);
275
276     for (b = 0; b < sb.size; b += BPB) {
277         bp = bread(dev, BBLOCK(b, sb.ninodes));
278
279         for (bi = 0; bi < BPB && b + bi < sb.size; bi++) {
280             m = 1 << (bi % 8);
281
282             if ((bp->data[bi / 8] & m) == 0) { // Is block free?
283                 bp->data[bi / 8] |= m; // Mark block in use.
284                 log_write(bp);
285                 brelse(bp);
286                 bzero(dev, b + bi);
287                 return b + bi;
288             }
289         }
290
291         brelse(bp);
292     }
293
294     panic("balloc:␣out␣of␣blocks");
295 }
296
297
298 static uint bmap (struct inode *ip, uint bn)
299 {
300     uint addr, *a;
301     struct buf *bp;
302
303     if (bn < NDIRECT) {
304         if ((addr = ip->addrs[bn]) == 0) {
305             ip->addrs[bn] = addr = balloc(ip->dev);
306         }
307
308         return addr;
309     }
310
311     bn -= NDIRECT;
312
313     if (bn < NINDIRECT) {
314         if ((addr = ip->addrs[NDIRECT]) == 0) {
315             ip->addrs[NDIRECT] = addr = balloc(ip->dev);
316         }
317
318         bp = bread(ip->dev, addr);
319         a = (uint*) bp->data;
320
321         if ((addr = a[bn]) == 0) {
322             a[bn] = addr = balloc(ip->dev);
323             log_write(bp);
324         }
325
326         brelse(bp);
327         return addr;
328     }
329
330     panic("bmap:␣out␣of␣range");
331 }
```

```
332
333
334 __code readi_loop(struct fs_impl* fs_impl, struct inode* ip, struct buf* bp, uint
        tot, uint m, char* dst, uint off, uint n, __code next(...)){ //:skip
335     bp = bread(ip->dev, bmap(ip, off / BSIZE));
336     m = min(n - tot, BSIZE - off%BSIZE);
337     memmove(dst, bp->data + off % BSIZE, m);
338     brelse(bp);
339     tot += m;
340     off += m;
341     dst += m;
342     goto readi_loopcheck(fs_impl, tot, m, dst, off, n, next(...));
343 }
344
345 __code readi_noloop(struct fs_impl* fs_impl, uint n, __code next(int ret, ...)){
346     ret = n;
347     goto next(ret, ...);
348 }
349
350 __code writei_check_diskinode(struct fs_impl* fs_impl,struct inode* ip, char* src,
        uint n, __code next(int ret, ...)){
351     if (ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write) {
352         ret = -1;
353         goto next(ret, ...);
354     }
355
356     ret = devsw[ip->major].write(ip, src, n);
357     goto next(ret, ...);
358 }
359
360 __code writei_loopcheck(struct fs_impl* fs_impl, uint tot, uint m, char* src, uint
        off, uint n, __code next(...)){
361     if(tot < n){
362         goto writei_loop(fs_impl, ip, bp, tot, m, src, off, n, next(...));
363     }
364     goto writei_noloop(fs_impl, next(...));
365 }
366
367 __code writei_loop(struct fs_impl* fs_impl, struct inode* ip, struct buf* bp, uint
        tot, uint m, char* src, uint off, uint n, __code next(...)){ //:skip
368     bp = bread(ip->dev, bmap(ip, off / BSIZE));
369     m = min(n - tot, BSIZE - off%BSIZE);
370     memmove(bp->data + off % BSIZE, src, m);
371     log_write(bp);
372     brelse(bp);
373     tot += m;
374     off += m;
375     src += m;
376     goto writei_loopcheck(fs_impl, tot, m, src, off, n, next(...));
377 }
378
379 __code writei_noloop(struct fs_impl* fs_impl, struct inode* ip, uint n, uint off,
        __code next(int ret, ...)){
380     if (n > 0 && off > ip->size) {
381         ip->size = off;
382         iupdate(ip);
383     }
384     ret = n;
385     goto next(ret, ...);
386 }
387 typedef struct dirent dirent;
388 __code dirlookup_loopcheck(struct fs_impl* fs_impl, struct inode* dp, char* name,
        uint off, uint* poff, dirent* de, __code next(...)){ //:skip
389     if(off < dp->size){
390         goto dirlookup_loop(fs_impl, dp, name, off, inum, poff, de, next(...));
391     }
```

```
392     goto dirlookup_noloop(fs_impl, next(...));
393 }
394
395 __code dirlookup_loop(struct fs_impl* fs_impl, struct inode* dp, char* name, uint
        off, uint inum, uint* poff, dirent* de, __code next(int ret, ...)){
396     if (readi(dp, (char*) &de, off, sizeof(de)) != sizeof(de)) {
397         char* msg = "dirlink␣read";
398         struct Err* err = createKernelError(&proc->cbc_context);
399         Gearef(cbc_context, Err)->msg = msg;
400         goto meta(cbc_context, err->panic);
401     }
402
403     if (de->inum == 0) {
404         off += sizeof(de);
405         goto dirlookup_loopcheck(fs_impl, dp, name, poff, de, next(...));
406     }
407
408     if (namecmp(name, de->name) == 0) {
409         if (poff) {
410             *poff = off;
411         }
412
413         inum = de->inum;
414         ret = iget(dp->dev, inum);
415         goto next(ret, ...);
416     }
417
418     off += sizeof(de);
419     goto dirlookup_loopcheck(fs_impl, dp, name, poff, de, next(...));
420 }
421
422 __code dirlookup_noloop(struct fs_impl* fs_impl, __code next(int ret, ...)){
423     ret = 0;
424     goto next(ret, ...);
425 }
426
427 __code dirlink_namecheck(struct fs_impl* fs_impl, struct inode* ip, __code next(int
        ret, ...)){
428     iput(ip);
429     ret = -1;
430     goto next(ret, ...);
431 }
432
433 __code dirlink_loopcheck(struct fs_impl* fs_impl, struct dirent* de, struct inode*
        dp, uint off, __code next(...)){ //:skip
434     if(off < dp->size){
435         goto dirlink_loop(fs_impl, de, dp, off, inum, next(...));
436     }
437     goto dirlink_noloop(fs_impl, de, dp, off, inum, name, next(...));
438 }
439
440 __code dirlink_loop(struct fs_impl* fs_impl, struct dirent* de, struct inode* dp,
        uint off, uint inum, __code next(...)){ //:skip
441     if (readi(dp, (char*) &de, off, sizeof(de)) != sizeof(de)) {
442        char* msg = "dirlink␣read";
443        struct Err* err = createKernelError(&proc->cbc_context);
444        Gearef(cbc_context, Err)->msg = msg;
445        goto meta(cbc_context, err->panic);
446     }
447
448     if (de->inum == 0) {
449          goto dirlink_noloop(fs_impl, de, dp, off, inum, name, next(...));
450     }
451
452     goto dirlink_loopcheck(fs_impl, de, dp, off + sizeof(de), next(...));
453 }
```

```
454
455 __code dirlink_noloop(struct fs_impl* fs_impl, struct dirent* de, struct inode* dp,
        uint off, uint inum, char* name, __code next(int ret, ...)){ //:skip
456     strncpy(de->name, name, DIRSIZ);
457     de->inum = inum;
458
459     if (writei(dp, (char*) &de, off, sizeof(de)) != sizeof(de)) {
460        char* msg = "dirlink␣read";
461        struct Err* err = createKernelError(&proc->cbc_context);
462        Gearef(cbc_context, Err)->msg = msg;
463        goto meta(cbc_context, err->panic);
464     }
465     ret = 0;
466     goto next(ret, ...);
467 }
```