

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 OS に対する信頼性の保証	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	6
3.1 GearsOS の概要	6
3.2 Context	6
3.3 Inetrface	7
第 4 章 xv6	8
4.1 xv6 の概要	8
4.2 xv6 の FileSystem	8
4.3 FileSystem の API	10
4.4 FileSystem の API の 相互関係	12
第 5 章 CbC による FileSystem の書き換え	13
5.1 書き換え方針	13
5.2 FileSystem の Interface の定義	13
5.3 FileSystem の Interface の実装	14
5.4 FileSystem の Interface の private な実装	17
第 6 章 まとめと今後の課題	21
参考文献	22
謝辞	23
発表履歴	24

付録 A ソースコード	25
A.1 FileSystem の Interface	25
A.2 fFileSystem Interface private 実装のヘッダーファイル	26
A.3 FileSystem Interface の 実装	27
A.4 FileSystem Interface の private 実装	33

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
2.5	stub CodeGear と DataGear の関係	5
3.1	CodeGear、DataGear、contxt の関係図	6
3.2	Stack の Interface とその実装	7
4.1	xv6 の FileSystem Layer	9
4.2	Disk 階層 のブロック割り当て	10
5.1	xv6 FileSystem の Interface と実装	13
5.2	allocinode の ループによる遷移図	20

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)	14
5.2	fs private のヘッダーファイル (fs_impl.h 一部抜粋)	14
5.3	FileSystem の Interface の実装 (fs_impl.cbc 一部抜粋)	14
5.4	iallocfs_impl の実装	17
5.5	iallocfs_impl の private 実装	17
A.1	FileSystem の Interface	25
A.2	fs private のヘッダーファイル	26
A.3	fs Interface の実装	27
A.4	fs Interface の private 実装	33

第1章 OS に対する信頼性の保証

OS には信頼性の保証と拡張性の実現が求められている。信頼性をノーマルレベルの計算に対して保証し、拡張性をメタレベルの計算で実現することを目標に Continuation based C[1](CbC) を用いて Gears OS[2] を開発中である。

CbC は Code Gear という基本的な処理の単位と Data Gear というデータの単位を用いる。細かい処理に対してノーマルレベルとメタレベルの Code Gear を記述し、stack に値を積む事なく CodeGear 間を遷移することができる。Code Gear に対して入力 of Data Gear と出力 of Data Gear が存在し、入力に対して期待される出力がされてるか検査することで信頼性を保証する。

CbC の Interface は Gears OS のモジュール化の仕組みである。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。Interface を使うことで検証や機能の入れ替えによる拡張が可能となる。

前段階としてシンプルであるがプロセス、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 OS である xv6 [3] を CbC で書き換えている。

本論文では、xv6 の FileSystem を CbC によって書き換えることにより複雑な処理である FileSystem を明確化させ信頼性を保証、Interface を使用可能とすることで拡張性を実現することを目標とする。

第2章 Continuation based C

2.1 Continuation based C の概要

Continuation based C (以下 CbC) は基本的な処理単位を CodeGear として定義し、CodeGear 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。

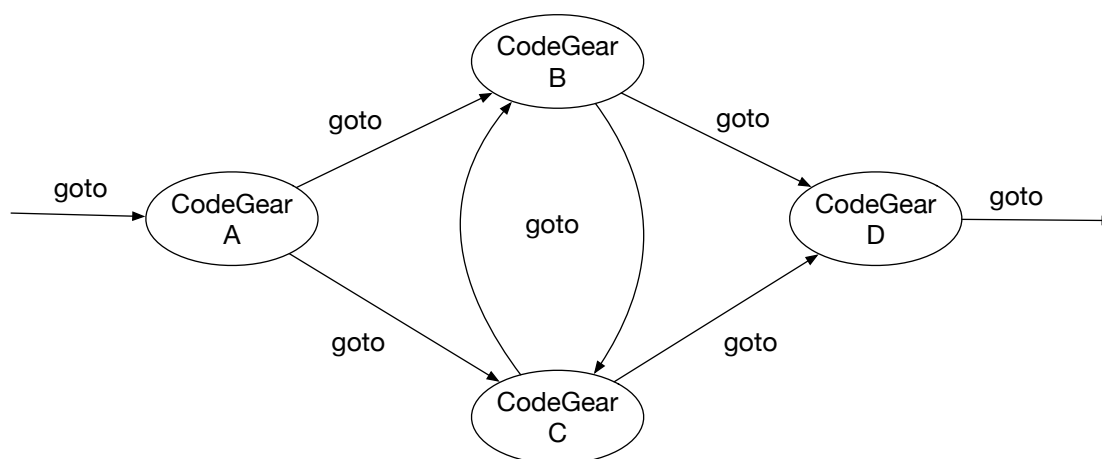


図 2.1: CodeGear 間の継続

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[4] [5] 及び LLVM[6] [7] 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 は戻り値を持たない為、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。

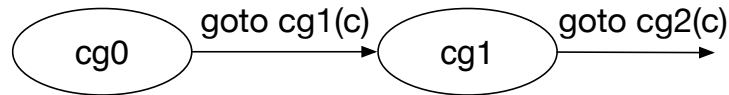


図 2.2: ソースコード 2.1 が表している CodeGear の状態遷移

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。

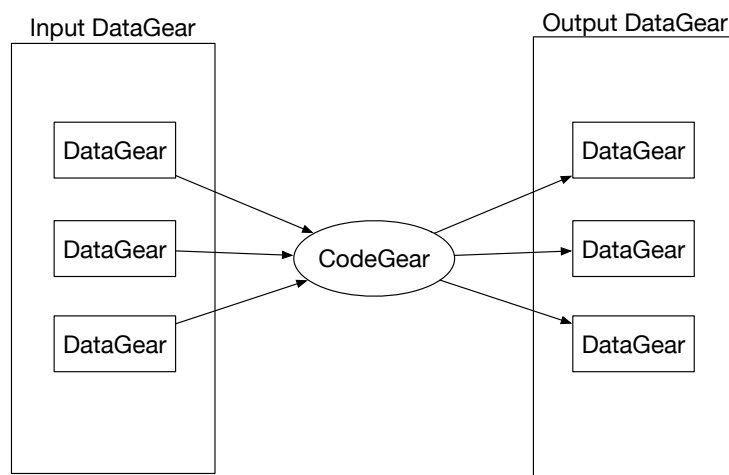


図 2.3: CodeGear と DataGear

2.4 Meta CodeGear と Meta DataGear

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

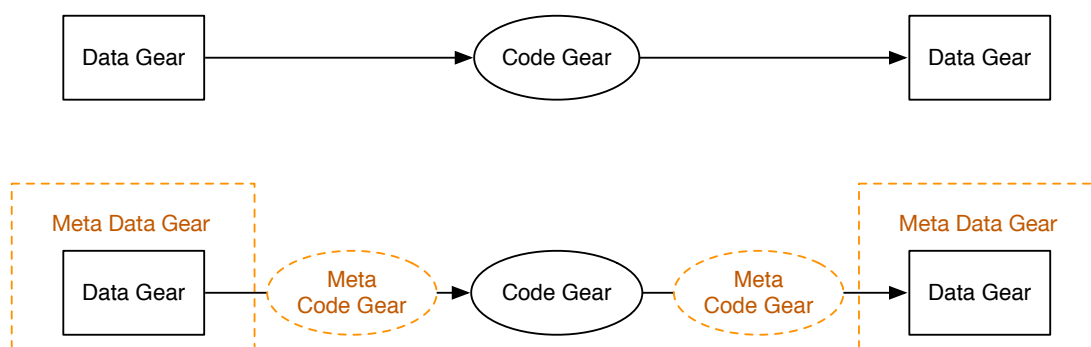


図 2.4: メタレベルの処理を可視化した CodeGear と DataGear

Meta DataGear は CbC の 接続可能な CodeGear、DataGear のリストであり DataGear 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。

Meta Code Gear の一種に stub CodeGear と呼ばれるものがある。stub CodeGear は Context から必要なデータを取り出して CodeGear に接続する Meta CodeGear であり、これを介して間接的に必要な DataGear にアクセスする。stub CodeGear と DataGear の関係について図 2.5 に示す。

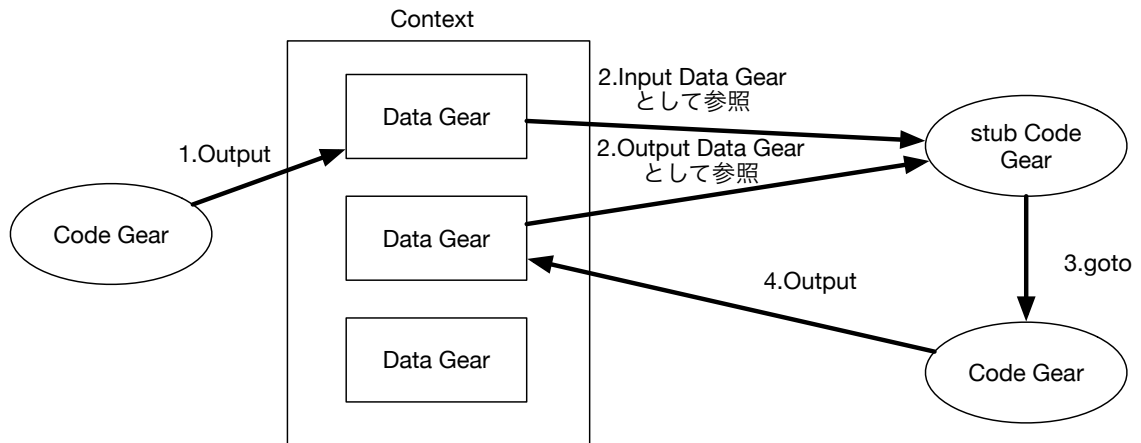


図 2.5: stub CodeGear と DataGear の関係

第3章 GearsOS

3.1 GearsOS の概要

Gears OS 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。

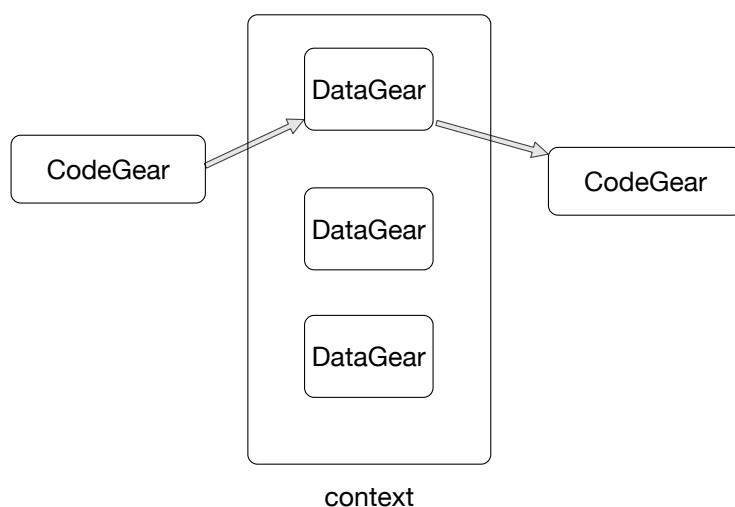


図 3.1: CodeGear、DataGear、context の関係図

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。

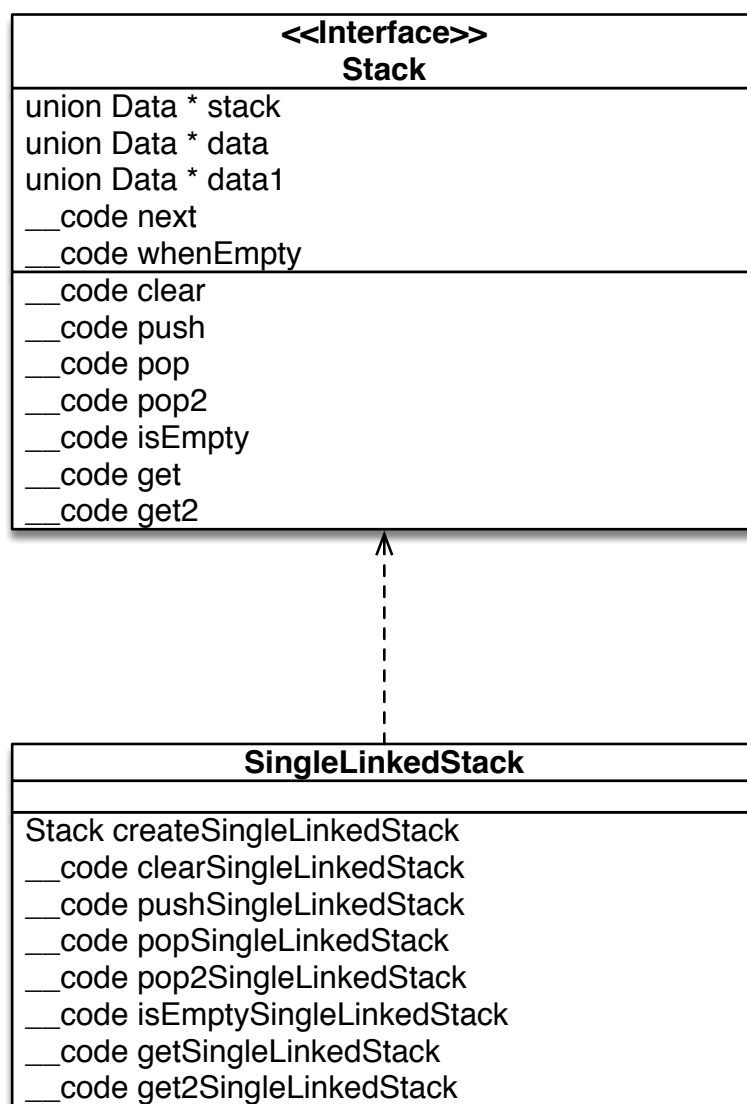


図 3.2: Stack の Interface とその実装

Interface の定義で示されている CodeGear と実装側で実装されている CodeGear は対応しており、実装側で実装する CodeGear は必ず Interface で定義されている必要がある。

第4章 xv6

4.1 xv6 の概要

xv6 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。

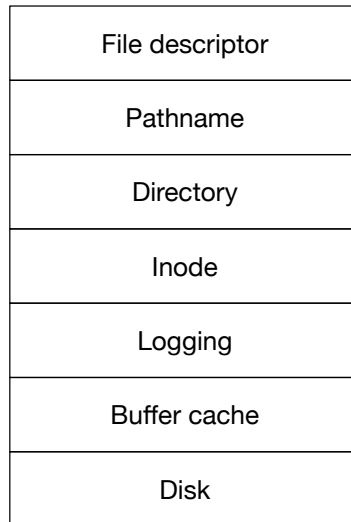


図 4.1: xv6 の FileSystem Layer

Disk 階層は、IDE ハードドライブ上のブロックを読み書きする。Buffer cache 階層はディスクブロックをキャッシュし、ブロックのアクセスの同期を取り、データが格納されている特定のブロックについて、複数のブロックが同時に変更を行っていないことを保証する。つまり、プロセス間の調停を行なっている。Logging 階層は高位の階層のいくつかのブロックの更新操作をトランザクションとしてラップし、またクラッシュが発生した場合でも、ブロックがシステムの他の部分から見てそれらがひとつの操作に見えるように更新されるようにする。inode 階層は個々のファイルを提供している。各ファイルは一意的な i-number とファイルのデータを保持するいくつかのブロックを持つ inode として表されている。Directory 階層はいくつかのディレクトリを構成している。ディレクトリの inode はディレクトリのエントリ列が含まれており、各エントリにはファイル名と i-number が入っている。Pathname 階層は、`/usr/rtn/xv6/fs.c` のような階層的なパスを構成している。File descriptor 階層は Unix の資源をファイルとして表現し、コンソールのようなデバイスはもちろん、実際のファイルもファイルとして表現している。

FileSystem は処理の際に Disk 階層のブロックを利用している。Disk 階層のブロックがどのように割り当てられているか図 4.2 に示す。

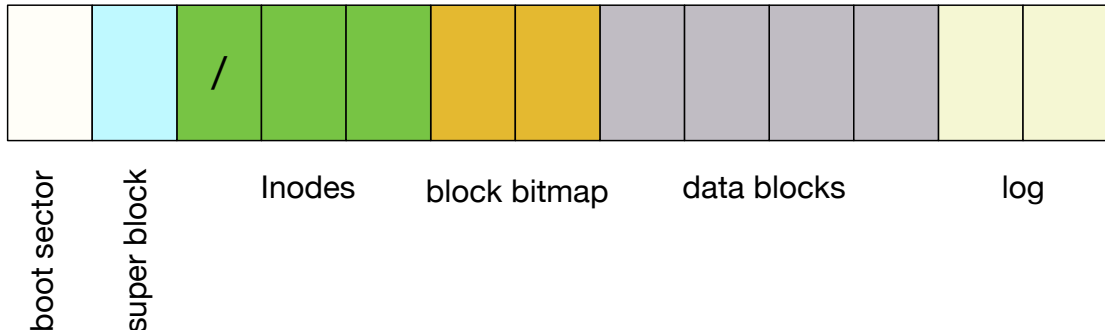


図 4.2: Disk 階層 のブロック割り当て

boot sector はシステムを起動するためのものであるが、FileSystem がこのブロックを使用することはない。super block は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。inodes は inode に関するリストが格納されている。block bitmap は使用しているブロックが記憶されている。data blocks は block bitmap において使用可能であることが記録されており、ファイルやディレクトリが保持されている。log は Logging 階層に関する情報が格納されている。

4.3 FileSystem の API

FileSystem について記述している fs.c ではファイル进行操作、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 Disk 階層の super block を読み込む。
- iinit
ブート時に main 関数で呼び出され初期化される。
- 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`
ディレクトリに関する文字列を比較している。ディレクトリを操作、管理する際に呼び出される。
- `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`
`/usr/rtm/xv6/fs.c` のような階層的なパスを構成する際の操作、管理時に呼び出される。
- `nameiparent`
`/usr/rtm/xv6/fs.c` のような階層的なパスを構成する際の操作、管理時に呼び出される。

4.4 FileSystem の API の 相互関係

iunlockput という iunlock と iput を組み合わせた API も存在する。さらに、ilock で inode をロックした場合必ず処理の終わりには iunlock が呼び出され inode のロックを解除する必要があるように、readi で inode からデータを読み込んだり iwrite で inode へデータを書き込む場合も iput で inode を参照必要がある。このように、FileSystem の API を利用する際 API の処理の中で他の API が呼ばれている場合もある。

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。FileSystem が記述されている fs.c で定義されている関数を CbC によって書き換えることにより処理の明確化と信頼性の保証をしたい。そのために今回は、Basic Block 単位に書き換えを行った。また、CbC の Interface を用いることにより仕様と実装を分離し拡張性を実現する。

5.2 FileSystem の Interface の定義

Interface 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 Gear と Data Gear の集合を表現していることに対し、Interface は一部の Code Gear と一部の Data Gear の集合を表現する。図 5.1 で表されている通り、FileSystem においても Interface で定義した CodeGear と実装側の CodeGear は対応している。

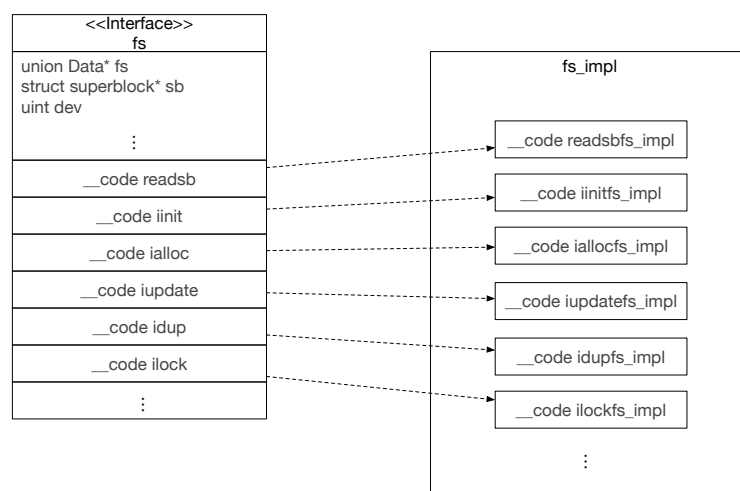


図 5.1: xv6 FileSystem の Interface と実装

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。

Interface の定義と同じように Interface の private な実装を定義したヘッダーファイルをソースコード 5.2 に記述する。

ソースコード 5.2: fs private のヘッダーファイル (fs_impl.h 一部抜粋)

```

1 typedef struct fs_impl<Type, Isa> impl fs{
2     __code allocinode(Type* fs_impl, uint dev, struct superblock* sb, __code next
3         (...));
4     __code allocinode_loop(Type* fs_impl, uint inum, uint dev, short type, struct
5         superblock* sb, struct buf* bp, struct dinode* dip, __code next(...));
6     __code allocinode_loopcheck(Type* fs_impl, uint inum, uint dev, struct superblock
7         * sb, struct buf* bp, struct dinode* dip, __code next(...));
8     __code allocinode_noloop(Type* fs_impl, uint inum, uint dev, short type, struct
9         superblock* sb, struct buf* bp, struct dinode* dip, __code next(int ret, ...)
10        );
11
12     ....
13 } fs_impl;

```

Interface の private な実装を記述する際に利用する CodeGear とソースコード 5.2 の CodeGear は Interface とその実装の時と同様に対応している。

5.3 FileSystem の Interface の実装

Interface の実装はソースコード 5.3 で示すように Interface とは別に定義する。

ソースコード 5.3: 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     initlock(&icache.lock, "icache");
84
85     goto next(...);
86 }
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     struct buf *bp;
95     struct dinode *dip;
96
97     bp = bread(ip->dev, IBLOCK(ip->inum));
98
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     acquire(&icache.lock);
116     ip->ref++;
117     release(&icache.lock);
118
119     goto next(ip, ...);
120 }
121
122
123 __code ilockfs_impl(struct fs_impl* fs, struct inode* ip, __code next(...)) {
124     goto lockinode1(fs, ip, bp, dip, next(...));
125 }
126
127
128 __code iunlockfs_impl(struct fs_impl* fs, struct inode* ip, __code next(...)) {
129     if (ip == 0 || !(ip->flags & I_BUSY) || ip->ref < 1) {
130         char* msg = "iunlock";
131         struct Err* err = createKernelError(&proc->cbc_context);
132         Gearef(cbc_context, Err)->msg = msg;
133         goto meta(cbc_context, err->panic);
134     }
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 Interface のヘッダファイルは `#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

Interface で定義した CodeGear 以外の CodeGear も記述することができる。この CodeGear は基本的に Interface で指定された CodeGear 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。

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 5.4 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

```

1 #interface "fs.dg"
2
3 __code iallocfs_impl(struct fs_impl* fs, uint dev, short type, __code next(...)) {
4     goto allocinode(fs, dev, sb, next(...));
5 }

```

`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

```

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) {
74         empty = ip;
75     }
76 }
77
78     if (empty == 0) {
79         panic("iget: 1no 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,
93     struct superblock* sb, struct buf* bp, struct dinode* dip, __code next(int ret,
94     ...)){ ///skip
95
96     memset(dip, 0, sizeof(*dip));
97     dip->type = type;
98     log_write(bp);
99     brelse(bp);
100
101     ret = iget(dev, inum);
102     goto next(ret, ...);

```

今回の実装は最初の遷移先である 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 へと処理が継続し停止する。static なものはまだ書き直していないが後々実装する。これらの CodeGear の一連の挙動を図 5.2 に示す。

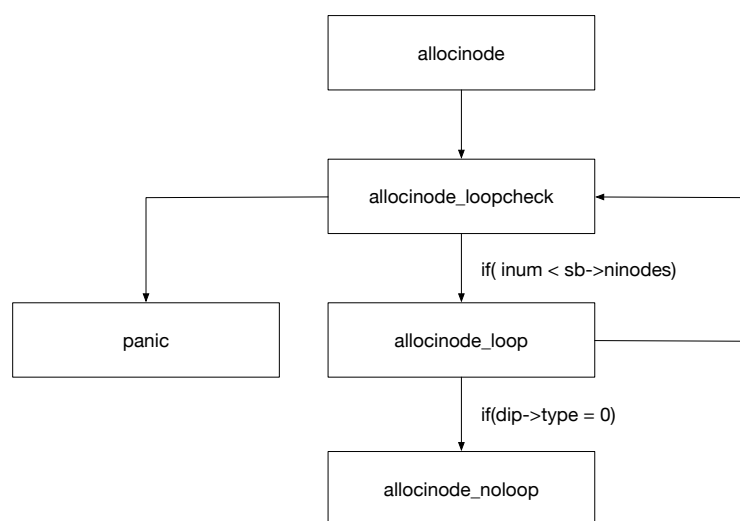


図 5.2: allocinode のループによる遷移図

本稿では `ialloc_impl` という関数を取り上げて説明したが、FileSystem が記述されている `fs.c` で定義されている関数に関しては全て CbC の Interface を用いることにより再実装した。

第6章 まとめと今後の課題

本研究では xv6 の FileSystem 部分について CbC を用いて書き換えを行った。実際に FileSystem を CbC で書き換えることによって、if 文と for 文を切り出してやるができた。さらに、FileSystem を Interface とその実装に書き換えることによって仕様と実装に分けることができた。仕様と実装に分けることによって拡張性を高めることができた。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 に移植しやすい形に改良していく必要がある。xv6 の FileSystem 部分書き換え後、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、正常に動作しなかった場合は修正を行い、OS として機能しているか再確認する必要がある。

今後の課題として、現段階では FileSystem と Paging などの kernel の一部のみ書き換えられているため kernel 全体の書き換えを完了させる必要がある。

参考文献

- [1] Kaito TOKKMORI and Shinji KONO. Implementing continuation based language in llvm and clang. *LOLA 2015*, July 2015.
- [2] 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.
- [3] 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).
- [4] GNU Compiler Collection (GCC) Internals. <http://gcc.gnu.org/onlinedocs/gccint/>.
- [5] 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.
- [6] 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.
- [7] Tokumori Kaito and Kono Shinji. Implementing continuation based language in llvm and clang. *LOLA 2015, Kyoto*, July 2015.
- [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りました河野真治准教授に深く感謝いたします。

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外間政尊さん、桃原優さん、清水隆博さん、東恩納琢偉さん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一木貴裕君、安田亮君、福田光希君に感謝致します。

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

発表履歴

- 坂本昂弘, 桃原優, 河野真治. 継続を用いた x. v6 kernel の書き換え. 研究報告システムソフトウェアとオペレーティング・システム (OS), April,2019

付録A ソースコード

本文中で紹介したソースコードの内、量が膨大なため一部しか掲載できなかったソースコードを示す。

A.1 FileSystem の Interface

FileSystem の Interface を定義したソースコードの全記述をソースコード A.1 に示す。

ソースコード A.1: FileSystem の Interface

```
1 typedef struct fs<Type,Impl> {
2     union Data* fs;
3     struct superblock* sb;
4     uint dev;
5     short type;
6     struct inode* ip;
7     struct stat* st;
8     char* dst;
9     uint off;
10    uint n;
11    const char* s;
12    const char* t;
13    struct inode* dp;
14    char* name;
15    uint* poff;
16    uint inum;
17    char* path;
18    char* src;
19    int namex_val;
20    int strncmp_val;
21    dirent* de;
22    int ret;
23    uint tot;
24    __code readsb(Impl* fs, uint dev, struct superblock* sb, __code next(...));
25    __code iinit(Impl* fs, __code next(...));
26    __code ialloc(Impl* fs, uint dev, short type, __code next(...));
27    __code iupdate(Impl* fs, struct inode* ip, __code next(...));
28    __code idup(Impl* fs, struct inode* ip, __code next(...));
29    __code ilock(Impl* fs, struct inode* ip, __code next(...));
30    __code iunlock(Impl* fs, struct inode* ip, __code next(...));
31    __code iput(Impl* fs, struct inode* ip, __code next(...));
32    __code iunlockput(Impl* fs, struct inode* ip, __code next(...));
33    __code stati(Impl* fs, struct inode* ip, struct stat* st, __code next(...));
34    __code readi(Impl* fs, struct inode* ip, char* dst, uint off, uint tot, uint n,
35                __code next(int ret, ...));
35    __code writei(Impl* fs, struct inode* ip, char* src, uint off, uint tot, uint n,
36                __code next(int ret, ...));
36    __code namecmp(Impl* fs, const char* s, const char* t, __code next(int
    strncmp_val, ...));
```

```

37  __code dirlookup(Impl* fs, struct inode* dp, char* name, uint off, uint* poff,
    dirent* de, __code next(int ret, ...));
38  __code dirlink(struct fs_impl* fs, struct inode* ip, struct dirent* de, struct
    inode* dp, char* name, uint off, uint inum, __code next(...));
39  __code namei(Impl* fs, char* path, __code next(int namex_val, ...));
40  __code nameiparent(Impl* fs, char* path, char* name, __code next(int namex_val,
    ...));
41  __code next(...);
42 } fs;

```

A.2 fFileSystem Interface private 実装のヘッダーファイル

FileSystem Interface の private 実装 に利用する全ての定義がされているヘッダーファイルの全ソースコードをソースコード A.2 に示す。

ソースコード A.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;

```

A.3 FileSystem Interface の実装

FileSystem Interface を実装したソースコードの全記述をソースコード A.3 に示す。

ソースコード A.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_readsdfs_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 readsdfs_impl(struct fs_impl* fs, uint dev, struct superblock* sb, __code
82     next(...)) { //:skip
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     initlock(&icache.lock, "icache");
99
100    goto next(...);
101 }
102
103 __code iallocfs_impl(struct fs_impl* fs, uint dev, short type, __code next(...)) {
104     goto allocinode(fs, dev, sb, next(...));
105 }
106
107 __code iupdatefs_impl(struct fs_impl* fs, struct inode* ip, __code next(...)) {
108     struct buf *bp;
109     struct dinode *dip;
110
111     bp = bread(ip->dev, IBLOCK(ip->inum));
112
113     dip = (struct dinode*) bp->data + ip->inum % IPB;
114     dip->type = ip->type;
115     dip->major = ip->major;
116     dip->minor = ip->minor;
117     dip->nlink = ip->nlink;
118     dip->size = ip->size;
119
120     memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
121     log_write(bp);
122     brelse(bp);
123
124     goto next(...);
125 }
126
127 __code idupfs_impl(struct fs_impl* fs, struct inode* ip, __code next(...)) {
128     acquire(&icache.lock);
129     ip->ref++;
130     release(&icache.lock);
131
132     goto next(ip, ...);
133 }
134
135 __code ilockfs_impl(struct fs_impl* fs, struct inode* ip, __code next(...)) {
136     goto lockinode1(fs, ip, bp, dip, next(...));
137 }
138
139 __code iunlockfs_impl(struct fs_impl* fs, struct inode* ip, __code next(...)) {
140     if (ip == 0 || !(ip->flags & I_BUSY) || ip->ref < 1) {
141         char* msg = "iunlock";
142         struct Err* err = createKernelError(&proc->cbc_context);
143         Gearef(cbc_context, Err)->msg = msg;
144         goto meta(cbc_context, err->panic);
145     }
146
147     acquire(&icache.lock);
148     ip->flags &= ~I_BUSY;
149     wakeup(ip);
150     release(&icache.lock);
151
152     goto next(...);
153 }

```

```

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
175     next(...)) { //:skip
176     st->dev = ip->dev;
177     st->ino = ip->inum;
178     st->type = ip->type;
179     st->nlink = ip->nlink;
180     st->size = ip->size;
181     goto next(...);
182 }
183
184 __code readifs_impl(struct fs_impl* fs, struct inode* ip, char* dst, uint off, uint
185     tot, uint n, __code next(int ret, ...)) {
186     if (ip->type == T_DEV) {
187         goto readi_check_diskinode(fs, ip, dst, n, next(...));
188     }
189     if (off > ip->size || off + n < off) {
190         ret = -1;
191         goto next(ret, ...);
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
201     tot, uint n, __code next(int ret, ...)) {
202     if (ip->type == T_DEV) {
203         goto writei_check_diskinode(fs, ip, src, n, next(...));
204     }
205     if (off > ip->size || off + n < off) {
206         ret = -1;
207         goto next(ret, ...);
208     }
209     if (off + n > MAXFILE * BSIZE) {
210         ret = -1;
211         goto next(ret, ...);
212     }
213     Gearef(cbc_context, fs)->tot = 0;
214     goto writei_loopcheck(fs, tot, m, src, off, n, next(...));
215 }
216
217
218
219 __code namecmpfs_impl(struct fs_impl* fs, const char* s, const char* t, __code next(
220     int strncmp_val, ...)) {
221     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,
225     uint* poff, dirent* de, __code next(...)) { //:skip
226     if (dp->type != T_DIR) {
227         char* msg = "dirlookup_not_DIR";
228         struct Err* err = createKernelError(&proc->cbc_context);
229         Gearef(cbc_context, Err)->msg = msg;
230         goto meta(cbc_context, err->panic);
231     }
232     Gearef(cbc_context, fs)->off = 0;
233     goto dirlookup_loopcheck(fs, dp, name, off, poff, de, next(...));
234 }
235
236 __code dirlinkfs_impl(struct fs_impl* fs, struct inode* ip, struct dirent* de,
237     struct inode* dp, char* name, uint off, uint inum, __code next(...)) { //:skip
238     if ((ip = dirlookup(dp, name, 0)) != 0) {
239         goto dirlink_namecheck(fs, ip, next(...));
240     }
241     Gearef(cbc_context, fs)->off = 0;
242     goto dirlink_loopcheck(fs, de, dp, off, next(...));
243 }
244
245 static struct inode* iget (uint dev, uint inum)
246 {
247     struct inode *ip, *empty;
248
249     acquire(&icache.lock);
250
251     empty = 0;
252
253     for (ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++) {
254         if (ip->ref > 0 && ip->dev == dev && ip->inum == inum) {
255             ip->ref++;
256             release(&icache.lock);
257             return ip;
258         }
259
260         if (empty == 0 && ip->ref == 0) { // Remember empty slot.
261             empty = ip;
262         }
263     }
264
265     if (empty == 0) {
266         panic("iget: no inodes");
267     }
268
269     ip = empty;
270     ip->dev = dev;
271     ip->inum = inum;
272     ip->ref = 1;
273     ip->flags = 0;
274     release(&icache.lock);
275
276     return ip;
277 }
278
279 static char* skipelem (char *path, char *name)
280 {
281     char *s;
282     int len;
283
284     while (*path == '/') {
285         path++;
286     }

```

```

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 }

```

A.4 FileSystem Interface の private 実装

FileSystem Interface の private 実装を記述した全ソースコードをソースコード A.4 に示す。

ソースコード A.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) {
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,
241     uint n, __code next(int ret, ...)){
242     if (ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read) {
243         ret = -1;
244         goto next(ret, ...);
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
251     off, uint n, __code next(...)){
252     if (tot < n){
253         goto readi_loop(fs_impl, ip, bp, tot, m, dst, off, n, next(...));
254     }
255     goto readi_noloop(fs_impl, next(...));
256 }
257
258 static void bzero (int dev, int bno)
259 {
260     struct buf *bp;
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) {
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     brelease(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
428     ret, ...)){
429     iput(ip);
430     ret = -1;
431     goto next(ret, ...);
432 }
433 __code dirlink_loopcheck(struct fs_impl* fs_impl, struct dirent* de, struct inode*
434     dp, uint off, __code next(...)){ //:skip
435     if(off < dp->size){
436         goto dirlink_loop(fs_impl, de, dp, off, inum, next(...));
437     }
438     goto dirlink_noloop(fs_impl, de, dp, off, inum, name, next(...));
439 }
440 __code dirlink_loop(struct fs_impl* fs_impl, struct dirent* de, struct inode* dp,
441     uint off, uint inum, __code next(...)){ //:skip
442     if (readi(dp, (char*) &de, off, sizeof(de)) != sizeof(de)) {
443         char* msg = "dirlink_read";
444         struct Err* err = createKernelError(&proc->cbc_context);
445         Gearef(cbc_context, Err)->msg = msg;
446         goto meta(cbc_context, err->panic);
447     }
448     if (de->inum == 0) {
449         goto dirlink_noloop(fs_impl, de, dp, off, inum, name, next(...));
450     }
451     goto dirlink_loopcheck(fs_impl, de, dp, off + sizeof(de), next(...));
452 }
453 }
454
455 __code dirlink_noloop(struct fs_impl* fs_impl, struct dirent* de, struct inode* dp,
456     uint off, uint inum, char* name, __code next(int ret, ...)){ //:skip
457     strncpy(de->name, name, DIRSIZ);
458     de->inum = inum;
459
460     if (writei(dp, (char*) &de, off, sizeof(de)) != sizeof(de)) {
461         char* msg = "dirlink_read";
462         struct Err* err = createKernelError(&proc->cbc_context);
463         Gearef(cbc_context, Err)->msg = msg;
464         goto meta(cbc_context, err->panic);
465     }
466     ret = 0;
467     goto next(ret, ...);
468 }

```