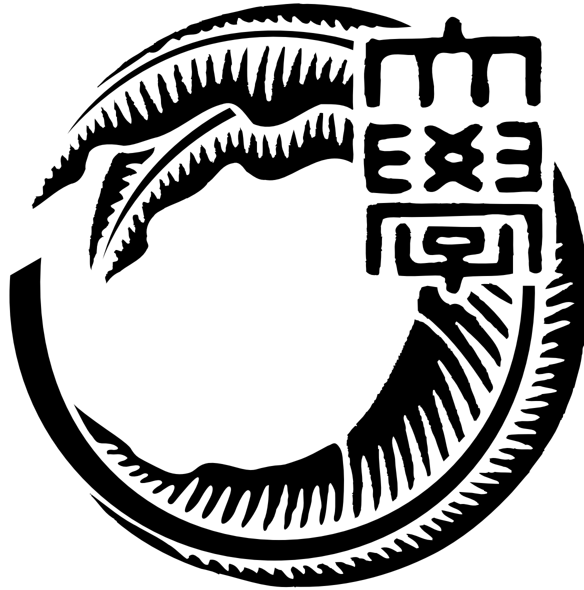
令和元年度 卒業論文

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

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 構造 7
4.3　FilleSystem の API 9

第 5 章　CbC による FileSystem の書き換え **11**
5.1　書き換え方針 11
5.2　FileSystem の Interface 11
5.3　CbC による FileSystem の書き換え 12

第 6 章　まとめと今後の課題 **13**

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

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 構造 7
4.2 xv6 の FileSystem に関する Disk の割り当て 9

表 目 次

ソースコード目次

2.1 CodeGear の継続の例 3
5.1 FileSystem の Interface 11
6.1 FileSystem の Interface 16
src/fs_impl.h 16
6.2 FileSystem の 実装 18
6.3 FileSystem の 実装 23

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.

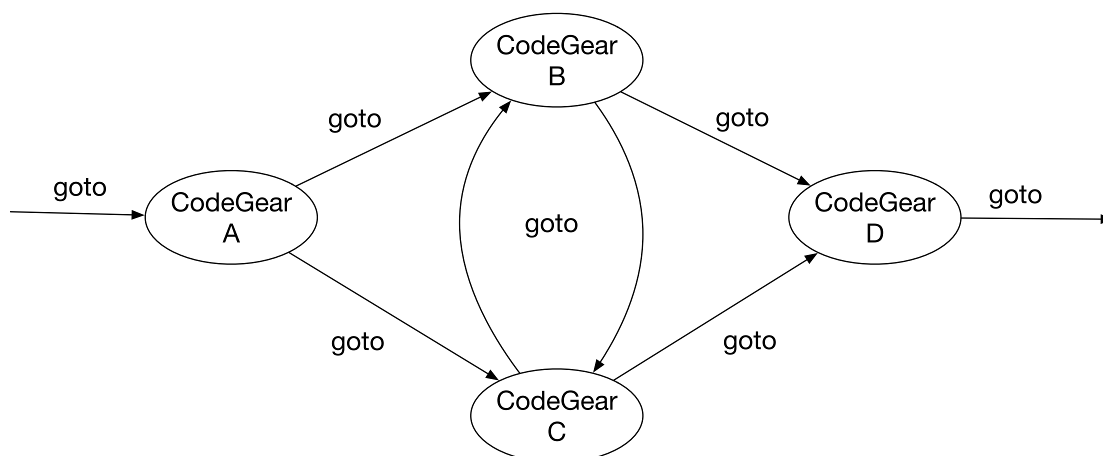


図 2.1: CodeGear 間の継続

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

```
__code cg0(Integer a, Integer b){
 int a_v = a->value;
 int b_v = b->value;
 Integer c = {a_v + b_v};
  goto cg1(c);
}
__code cg1(Integer c){
  goto cg2(c);
}
```

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.

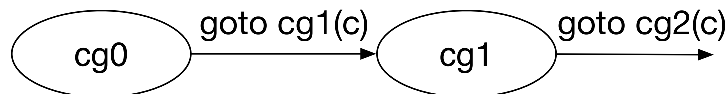


図 2.2: ソースコード 2.1 が表している CodeGear の状態遷移

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で示した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.

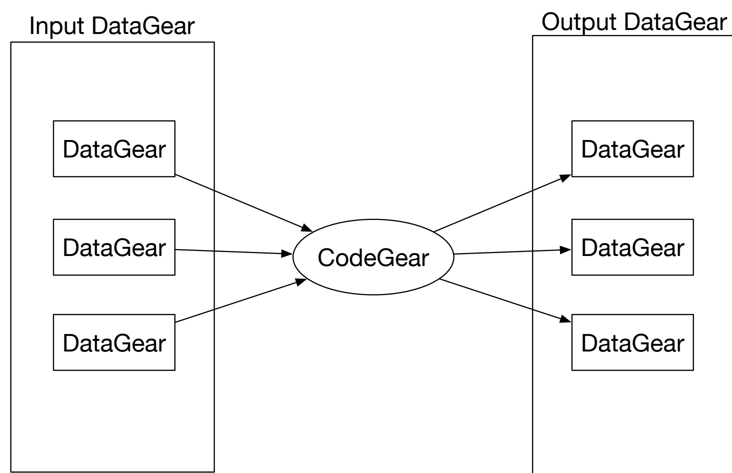


図 2.3: CodeGear と DataGear

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 を全て持っている 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 CodeGar,DataGer および context の関係を以下の図 3.1 に簡潔に示す.

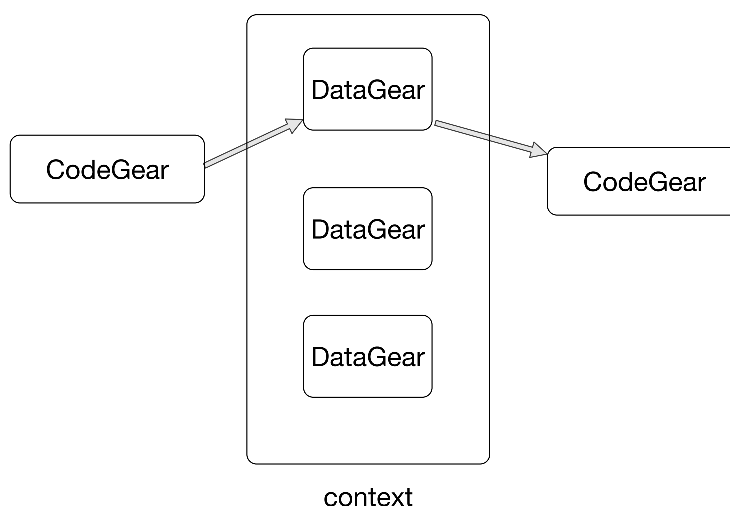


図 3.1: CodeGear,DataGear,contxt の関係図

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.

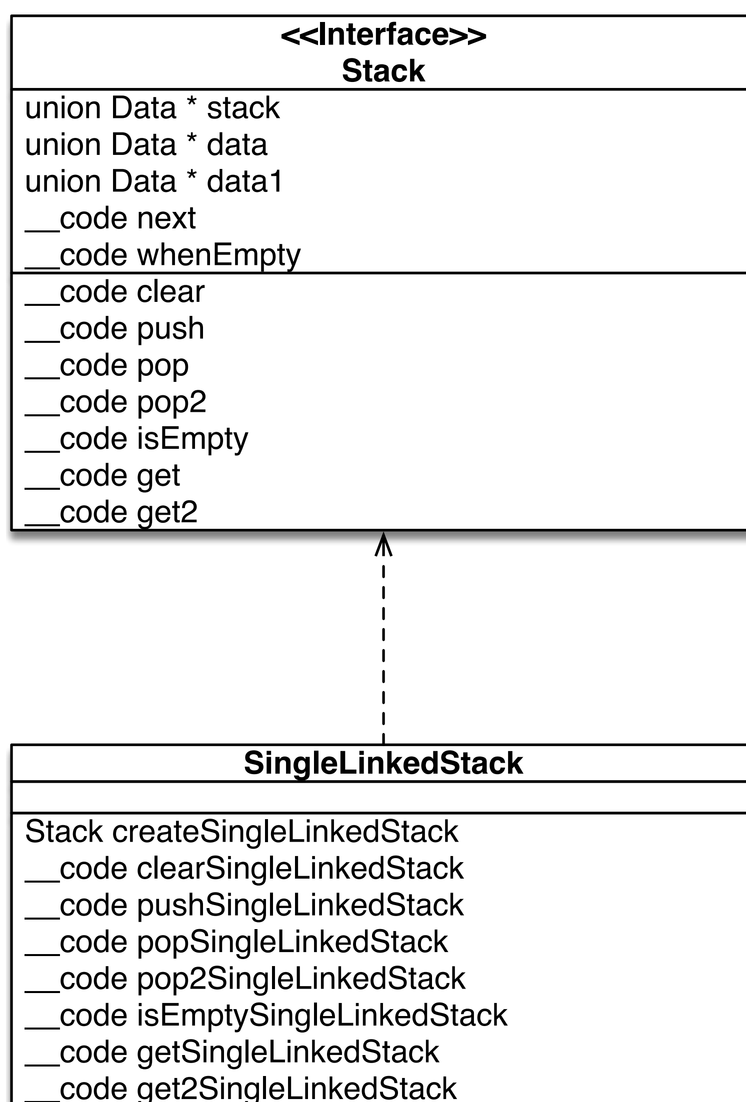


図 3.2: Stack の Interface とその実装

第4章　xv6

4.1　xv6 の概要

xv6 [6] 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[7] 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 構造

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.

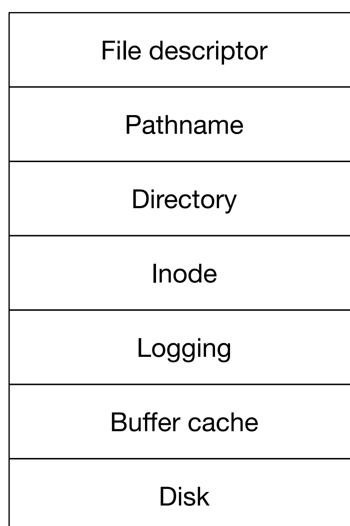


図 4.1: xv6 の FileSystem 構造

- File descriptor 階層 Unix の資源はファイルとして表現され, 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されている. File descriptor 階層はこの

- Pathname 階層
- Directory 階層
- Inode 階層
- Logging 階層
- Bffer cache 階層
- Disk 階層

xv6 の FileSystem の Disk の割り当てを以下の図 4.2 に示す.

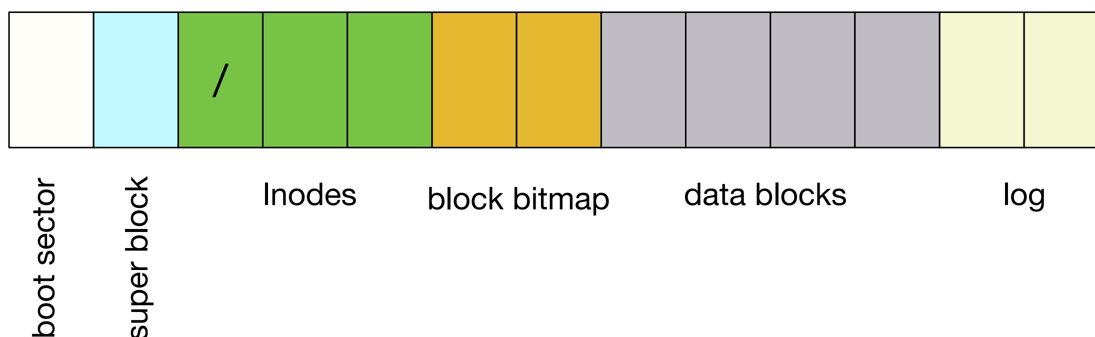


図 4.2: xv6 の FileSystem に関する Disk の割り当て

- boot sector
  boot sector を保持しているだけで FileSystem はこのブロックを使用することはない.
- super block
  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
  inode が格納されている.
- block bitmap
  block bitmap は使用しているブロックが記憶されている.
- data blocks
  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
  Logging 階層の log が格納されている.

4.3 FilleSystem の API

-
-
-
-

-
-
-

第5章　CbCによるFileSystemの書き換え

5.1　書き換え方針

5.2　FileSystemのInterface

ソースコード 5.1: 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, ...));
```

```
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;
```

5.3 CbC による FileSystem の書き換え

第6章　まとめと今後の課題

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] 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).

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

[8] 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.

[9] 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;
```

```
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 6.2: FileSystem 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 // ----
15 // typedef struct fs_impl<Impl, Isa> impl fs{
16 // union Data* fs_impl;
17 //
18 //
19 //
20 //
21 // } fs_impl;
22 // ----
23
24 fs* createfs_impl(struct Context* cbc_context) {
25     struct fs* fs = new fs();
26     struct fs_impl* fs_impl = new fs_impl();
27     fs->fs = (union Data*)fs_impl;
28     fs_impl->fs_impl = NULL;
29     fs_impl->sb = NULL;
30     fs_impl->ret = 0;
31     fs_impl->dev = 0;
32     fs_impl->type = 0;
33     fs_impl->bp = NULL;
34     fs_impl->dip = NULL;
35     fs_impl->inum = 0;
36     fs_impl->dp = NULL;
37     fs_impl->name = NULL;
38     fs_impl->off = 0;
39     fs_impl->poff = NULL;
40     fs_impl->de = NULL;
41     fs_impl->tot = 0;
42     fs_impl->m = 0;
43     fs_impl->dst = NULL;
44     fs_impl->n = 0;
45     fs_impl->src = NULL;
46     fs_impl->allocinode = C_allocinode;
47     fs_impl->allocinode_loop = C_allocinode_loop;
48     fs_impl->allocinode_loopcheck = C_allocinode_loopcheck;
49     fs_impl->allocinode_noloop = C_allocinode_noloop;
50     fs_impl->lockinode1 = C_lockinode1;
51     fs_impl->lockinode2 = C_lockinode2;
52     fs_impl->lockinode_sleepcheck = C_lockinode_sleepcheck;
53     fs_impl->iput_check = C_iput_check;
54     fs_impl->iput_inode_nolink = C_iput_inode_nolink;
55     fs_impl->readi_check_diskinode = C_readi_check_diskinode;
56     fs_impl->readi_loopcheck = C_readi_loopcheck;
57     fs_impl->readi_loop = C_readi_loop;
58     fs_impl->readi_noloop = C_readi_noloop;
59     fs_impl->writei_check_diskinode = C_writei_check_diskinode;
60     fs_impl->writei_loopcheck = C_writei_loopcheck;
61     fs_impl->writei_loop = C_writei_loop;
```

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

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

```
192     goto next(...);
193 }
194
195 __code readifs_impl(struct fs_impl* fs, struct inode* ip, char* dst, uint off, uint
        tot, uint n, __code next(int ret, ...)) {
196     if (ip->type == T_DEV) {
197         goto readi_check_diskinode(fs, ip, dst, n, next(...));
198     }
199
200     if (off > ip->size || off + n < off) {
201         ret = -1;
202         goto next(ret, ...);
203     }
204
205     if (off + n > ip->size) {
206         n = ip->size - off;
207     }
208     Gearef(cbc_context, fs)->tot = 0;
209     goto readi_loopcheck(fs, tot, m, dst, off, n, next(...));
210 }
211
212 __code writeifs_impl(struct fs_impl* fs, struct inode* ip, char* src, uint off, uint
         tot, uint n, __code next(int ret, ...)) {
213     if (ip->type == T_DEV) {
214         goto writei_check_diskinode(fs, ip, src, n, next(...));
215     }
216
217     if (off > ip->size || off + n < off) {
218         ret = -1;
219         goto next(ret, ...);
220     }
221
222     if (off + n > MAXFILE * BSIZE) {
223         ret = -1;
224         goto next(ret, ...);
225     }
226     Gearef(cbc_context, fs)->tot = 0;
227     goto writei_loopcheck(fs, tot, m, src, off, n, next(...));
228 }
229
230
231 __code namecmpfs_impl(struct fs_impl* fs, const char* s, const char* t, __code next(
        int strncmp_val, ...)) {
232     strncmp_val = strncmp(s, t, DIRSIZ);
233     goto next(strncmp_val, ...);
234 }
235
236 __code dirlookupfs_impl(struct fs_impl* fs, struct inode* dp, char* name, uint off,
        uint* poff, dirent* de, __code next(...)) { //:skip
237     if (dp->type != T_DIR) {
238        char* msg = "dirlookup␣not␣DIR";
239        struct Err* err = createKernelError(&proc->cbc_context);
240        Gearef(cbc_context, Err)->msg = msg;
241        goto meta(cbc_context, err->panic);
242     }
243     Gearef(cbc_context, fs)->off = 0;
244     goto dirlookup_loopcheck(fs, dp, name, off, poff, de, next(...));
245 }
246
247 __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
248     // Check that name is not present.
249     if ((ip = dirlookup(dp, name, 0)) != 0) {
250         goto dirlink_namecheck(fs, ip, next(...));
251     }
252     Gearef(cbc_context, fs)->off = 0;
```

```
253     goto dirlink_loopcheck(fs, de, dp, off, next(...));
254 }
255
256 static struct inode* iget (uint dev, uint inum)
257 {
258     struct inode *ip, *empty;
259
260     acquire(&icache.lock);
261
262     // Is the inode already cached?
263     empty = 0;
264
265     for (ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++) {
266         if (ip->ref > 0 && ip->dev == dev && ip->inum == inum) {
267             ip->ref++;
268             release(&icache.lock);
269             return ip;
270         }
271
272         if (empty == 0 && ip->ref == 0) { // Remember empty slot.
273             empty = ip;
274         }
275     }
276
277     // Recycle an inode cache entry.
278     if (empty == 0) {
279         panic("iget:␣no␣inodes");
280     }
281
282     ip = empty;
283     ip->dev = dev;
284     ip->inum = inum;
285     ip->ref = 1;
286     ip->flags = 0;
287     release(&icache.lock);
288
289     return ip;
290 }
291
292 static char* skipelem (char *path, char *name)
293 {
294     char *s;
295     int len;
296
297     while (*path == '/') {
298         path++;
299     }
300
301     if (*path == 0) {
302         return 0;
303     }
304
305     s = path;
306
307     while (*path != '/' && *path != 0) {
308         path++;
309     }
310
311     len = path - s;
312
313     if (len >= DIRSIZ) {
314         memmove(name, s, DIRSIZ);
315     } else {
316         memmove(name, s, len);
317         name[len] = 0;
318     }
```

```
319
320     while (*path == '/') {
321         path++;
322     }
323
324     return path;
325 }
326
327
328 static struct inode* namex (char *path, int nameiparent, char *name)
329 {
330     struct inode *ip, *next;
331
332     if (*path == '/') {
333         ip = iget(ROOTDEV, ROOTINO);
334     } else {
335         ip = idup(proc->cwd);
336     }
337
338     while ((path = skipelem(path, name)) != 0) {
339         ilock(ip);
340
341         if (ip->type != T_DIR) {
342             iunlockput(ip);
343             return 0;
344         }
345
346         if (nameiparent && *path == '\0') {
347             // Stop one level early.
348             iunlock(ip);
349             return ip;
350         }
351
352         if ((next = dirlookup(ip, name, 0)) == 0) {
353             iunlockput(ip);
354             return 0;
355         }
356
357         iunlockput(ip);
358         ip = next;
359     }
360
361     if (nameiparent) {
362         iput(ip);
363         return 0;
364     }
365
366     return ip;
367 }
368
369 __code nameifs_impl(struct fs_impl* fs, char* path, __code next(int namex_val, ...))
         {
370     char name[DIRSIZ];
371     namex_val = namex(path, 0, name);
372     goto next(namex_val, ...);
373 }
374
375 __code nameiparentfs_impl(struct fs_impl* fs, char* path, char* name, __code next(
        int namex_val, ...)) {
376
377     namex_val = namex(path, 1, name);
378     goto next(namex_val, ...);
379
380 }
```

ソースコード 6.3: FileSystem 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 "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      // Is the inode already cached?
65      empty = 0;
66
67      for (ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++) {
68          if (ip->ref > 0 && ip->dev == dev && ip->inum == inum) {
69              ip->ref++;
70              release(&icache.lock);
71              return ip;
72          }
73
74          if (empty == 0 && ip->ref == 0) { // Remember empty slot.
75              empty = ip;
76          }
77      }
78
79      // Recycle an inode cache entry.
80      if (empty == 0) {
81          panic("iget:␣no␣inodes");
82      }
83
84      ip = empty;
85      ip->dev = dev;
86      ip->inum = inum;
87      ip->ref = 1;
88      ip->flags = 0;
89      release(&icache.lock);
90
91      return ip;
92  }
93
94  __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
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, ...);
103
104 }
105
106 __code lockinode1(struct fs_impl* fs_impl, struct inode* ip, struct buf* bp, struct
        dinode* dip, __code next(...)){ //:skip
107
108     if (ip == 0 || ip->ref < 1) {
109        char* msg = "ilock";
110        struct Err* err = createKernelError(&proc->cbc_context);
111        Gearef(cbc_context, Err)->msg = msg;
112        goto meta(cbc_context, err->panic);
113      }
114     acquire(&icache.lock);
115
116     goto lockinode_sleepcheck(fs_impl, ip, next(...));
117
118 }
119
120
121 __code lockinode2(struct fs_impl* fs_impl, struct inode* ip, struct buf* bp, struct
        dinode* dip, __code next(...)){ //:skip
122
```

```
123     ip->flags |= I_BUSY;
124     release(&icache.lock);
125
126     if (!(ip->flags & I_VALID)) {
127         bp = bread(ip->dev, IBLOCK(ip->inum));
128
129         dip = (struct dinode*) bp->data + ip->inum % IPB;
130         ip->type = dip->type;
131         ip->major = dip->major;
132         ip->minor = dip->minor;
133         ip->nlink = dip->nlink;
134         ip->size = dip->size;
135
136         memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
137         brelse(bp);
138         ip->flags |= I_VALID;
139
140         if (ip->type == 0) {
141              char* msg = "ilock:␣no␣type";
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     goto next(...);
148 }
149 __code lockinode_sleepcheck(struct fs_impl* fs_impl, struct inode* ip, __code next
        (...)){
150     if(ip->flags & I_BUSY){
151         sleep(ip, &icache.lock);
152         goto lockinode_sleepcheck(fs_impl, ip, next(...));
153     }
154     goto lockinode2(fs_impl, ip, bp, dip, next(...));
155 }
156
157 __code iput_check(struct fs_impl* fs_impl, struct inode* ip, __code next(...)){
158     acquire(&icache.lock);
159     if (ip->ref == 1 && (ip->flags & I_VALID) && ip->nlink == 0) {
160         goto iput_inode_nolink(fs_impl, ip, next(...));
161     }
162     ip->ref--;
163     release(&icache.lock);
164     goto next(...);
165
166 }
167
168 static void bfree (int dev, uint b)
169 {
170     struct buf *bp;
171     struct superblock sb;
172     int bi, m;
173
174     readsb(dev, &sb);
175     bp = bread(dev, BBLOCK(b, sb.ninodes));
176     bi = b % BPB;
177     m = 1 << (bi % 8);
178
179     if ((bp->data[bi / 8] & m) == 0) {
180         panic("freeing␣free␣block");
181     }
182
183     bp->data[bi / 8] &= ~m;
184     log_write(bp);
185     brelse(bp);
186 }
187
```

```
188
189 static void itrunc (struct inode *ip)
190 {
191     int i, j;
192     struct buf *bp;
193     uint *a;
194
195     for (i = 0; i < NDIRECT; i++) {
196         if (ip->addrs[i]) {
197             bfree(ip->dev, ip->addrs[i]);
198             ip->addrs[i] = 0;
199         }
200     }
201
202     if (ip->addrs[NDIRECT]) {
203         bp = bread(ip->dev, ip->addrs[NDIRECT]);
204         a = (uint*) bp->data;
205
206         for (j = 0; j < NINDIRECT; j++) {
207             if (a[j]) {
208                 bfree(ip->dev, a[j]);
209             }
210         }
211
212         brelse(bp);
213         bfree(ip->dev, ip->addrs[NDIRECT]);
214         ip->addrs[NDIRECT] = 0;
215     }
216
217     ip->size = 0;
218     iupdate(ip);
219 }
220
221 __code iput_inode_nolink(struct fs_impl* fs_impl, struct inode* ip, __code next(...)
    ){
222
223     if (ip->flags & I_BUSY) {
224         char* msg = "iput␣busy";
225         struct Err* err = createKernelError(&proc->cbc_context);
226         Gearef(cbc_context, Err)->msg = msg;
227         goto meta(cbc_context, err->panic);
228     }
229
230     ip->flags |= I_BUSY;
231     release(&icache.lock);
232     itrunc(ip);
233     ip->type = 0;
234     iupdate(ip);
235
236     acquire(&icache.lock);
237     ip->flags = 0;
238     wakeup(ip);
239     goto next(...);
240 }
241
242 __code readi_check_diskinode(struct fs_impl* fs_impl,struct inode* ip, char* dst,
     uint n, __code next(int ret, ...)){
243     if (ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read) {
244         ret = -1;
245         goto next(ret, ...);
246     }
247
248     ret = devsw[ip->major].read(ip, dst, n);
249     goto next(ret, ...);
250 }
251
```

```
252 __code readi_loopcheck(struct fs_impl* fs_impl, uint tot, uint m, char* dst, uint
        off, uint n, __code next(...)){
253     if(tot < n){
254         goto readi_loop(fs_impl, ip, bp, tot, m, dst, off, n, next(...));
255     }
256     goto readi_noloop(fs_impl, next(...));
257 }
258 
259 static void bzero (int dev, int bno)
260 {
261     struct buf *bp;
262 
263     bp = bread(dev, bno);
264     memset(bp->data, 0, BSIZE);
265     log_write(bp);
266     brelse(bp);
267 }
268 
269 static uint balloc (uint dev)
270 {
271     int b, bi, m;
272     struct buf *bp;
273     struct superblock sb;
274 
275     bp = 0;
276     readsb(dev, &sb);
277 
278     for (b = 0; b < sb.size; b += BPB) {
279         bp = bread(dev, BBLOCK(b, sb.ninodes));
280 
281         for (bi = 0; bi < BPB && b + bi < sb.size; bi++) {
282             m = 1 << (bi % 8);
283 
284             if ((bp->data[bi / 8] & m) == 0) { // Is block free?
285                 bp->data[bi / 8] |= m; // Mark block in use.
286                 log_write(bp);
287                 brelse(bp);
288                 bzero(dev, b + bi);
289                 return b + bi;
290             }
291         }
292 
293         brelse(bp);
294     }
295 
296     panic("balloc: out of blocks");
297 }
298 
299 
300 static uint bmap (struct inode *ip, uint bn)
301 {
302     uint addr, *a;
303     struct buf *bp;
304 
305     if (bn < NDIRECT) {
306         if ((addr = ip->addrs[bn]) == 0) {
307             ip->addrs[bn] = addr = balloc(ip->dev);
308         }
309 
310         return addr;
311     }
312 
313     bn -= NDIRECT;
314 
315     if (bn < NINDIRECT) {
316         // Load indirect block, allocating if necessary.
```

```
317         if ((addr = ip->addrs[NDIRECT]) == 0) {
318             ip->addrs[NDIRECT] = addr = balloc(ip->dev);
319         }
320
321         bp = bread(ip->dev, addr);
322         a = (uint*) bp->data;
323
324         if ((addr = a[bn]) == 0) {
325             a[bn] = addr = balloc(ip->dev);
326             log_write(bp);
327         }
328
329         brelse(bp);
330         return addr;
331     }
332
333     panic("bmap:␣out␣of␣range");
334 }
335
336
337 __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
338     bp = bread(ip->dev, bmap(ip, off / BSIZE));
339     m = min(n - tot, BSIZE - off%BSIZE);
340     memmove(dst, bp->data + off % BSIZE, m);
341     brelse(bp);
342     tot += m;
343     off += m;
344     dst += m;
345     goto readi_loopcheck(fs_impl, tot, m, dst, off, n, next(...));
346 }
347
348 __code readi_noloop(struct fs_impl* fs_impl, uint n, __code next(int ret, ...)){
349     ret = n;
350     goto next(ret, ...);
351 }
352
353 __code writei_check_diskinode(struct fs_impl* fs_impl,struct inode* ip, char* src,
        uint n, __code next(int ret, ...)){
354     if (ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write) {
355         ret = -1;
356         goto next(ret, ...);
357     }
358
359     ret = devsw[ip->major].write(ip, src, n);
360     goto next(ret, ...);
361 }
362
363 __code writei_loopcheck(struct fs_impl* fs_impl, uint tot, uint m, char* src, uint
        off, uint n, __code next(...)){
364     if(tot < n){
365         goto writei_loop(fs_impl, ip, bp, tot, m, src, off, n, next(...));
366     }
367     goto writei_noloop(fs_impl, next(...));
368 }
369
370 __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
371     bp = bread(ip->dev, bmap(ip, off / BSIZE));
372     m = min(n - tot, BSIZE - off%BSIZE);
373     memmove(bp->data + off % BSIZE, src, m);
374     log_write(bp);
375     brelse(bp);
376     tot += m;
377     off += m;
378     src += m;
```

```
379     goto writei_loopcheck(fs_impl, tot, m, src, off, n, next(...));
380 }
381
382 __code writei_noloop(struct fs_impl* fs_impl, struct inode* ip, uint n, uint off,
        __code next(int ret, ...)){
383     if (n > 0 && off > ip->size) {
384         ip->size = off;
385         iupdate(ip);
386     }
387     ret = n;
388     goto next(ret, ...);
389 }
390 typedef struct dirent dirent;
391 __code dirlookup_loopcheck(struct fs_impl* fs_impl, struct inode* dp, char* name,
        uint off, uint* poff, dirent* de, __code next(...)){ //:skip
392     if(off < dp->size){
393         goto dirlookup_loop(fs_impl, dp, name, off, inum, poff, de, next(...));
394     }
395     goto dirlookup_noloop(fs_impl, next(...));
396 }
397
398 __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, ...)){
399     if (readi(dp, (char*) &de, off, sizeof(de)) != sizeof(de)) {
400         char* msg = "dirlink␣read";
401         struct Err* err = createKernelError(&proc->cbc_context);
402         Gearef(cbc_context, Err)->msg = msg;
403         goto meta(cbc_context, err->panic);
404     }
405
406     if (de->inum == 0) {
407         off += sizeof(de);
408         goto dirlookup_loopcheck(fs_impl, dp, name, poff, de, next(...));
409     }
410
411     if (namecmp(name, de->name) == 0) {
412         // entry matches path element
413         if (poff) {
414             *poff = off;
415         }
416
417         inum = de->inum;
418         ret = iget(dp->dev, inum);
419         goto next(ret, ...);
420     }
421
422     off += sizeof(de);
423     goto dirlookup_loopcheck(fs_impl, dp, name, poff, de, next(...));
424 }
425
426 __code dirlookup_noloop(struct fs_impl* fs_impl, __code next(int ret, ...)){
427     ret = 0;
428     goto next(ret, ...);
429 }
430
431 __code dirlink_namecheck(struct fs_impl* fs_impl, struct inode* ip, __code next(int
        ret, ...)){
432     iput(ip);
433     ret = -1;
434     goto next(ret, ...);
435 }
436
437 __code dirlink_loopcheck(struct fs_impl* fs_impl, struct dirent* de, struct inode*
        dp, uint off, __code next(...)){ //:skip
438     if(off < dp->size){
439         goto dirlink_loop(fs_impl, de, dp, off, inum, next(...));
```

```
440     }
441     goto dirlink_noloop(fs_impl, de, dp, off, inum, name, next(...));
442 }
443
444 __code dirlink_loop(struct fs_impl* fs_impl, struct dirent* de, struct inode* dp,
        uint off, uint inum, __code next(...)){ //:skip
445     if (readi(dp, (char*) &de, off, sizeof(de)) != sizeof(de)) {
446         char* msg = "dirlink␣read";
447         struct Err* err = createKernelError(&proc->cbc_context);
448         Gearef(cbc_context, Err)->msg = msg;
449         goto meta(cbc_context, err->panic);
450     }
451
452     if (de->inum == 0) {
453          goto dirlink_noloop(fs_impl, de, dp, off, inum, name, next(...));
454     }
455
456     goto dirlink_loopcheck(fs_impl, de, dp, off + sizeof(de), next(...));
457 }
458
459 __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
460     strncpy(de->name, name, DIRSIZ);
461     de->inum = inum;
462
463     if (writei(dp, (char*) &de, off, sizeof(de)) != sizeof(de)) {
464         char* msg = "dirlink␣read";
465         struct Err* err = createKernelError(&proc->cbc_context);
466         Gearef(cbc_context, Err)->msg = msg;
467         goto meta(cbc_context, err->panic);
468     }
469     ret = 0;
470     goto next(ret, ...);
471 }
```