令和元年度 卒業論文

CbCによるxv6のFileSystemの書き換え

琉球大学工学部情報工学科

165723C 坂本昂弘

指導教員 河野真治

目 次

図 目 次

表 目 次

ソースコード目次

第1章　はじめに

1.1　背景と目的

1.2　論文の構成

第2章　Continuation based C

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.

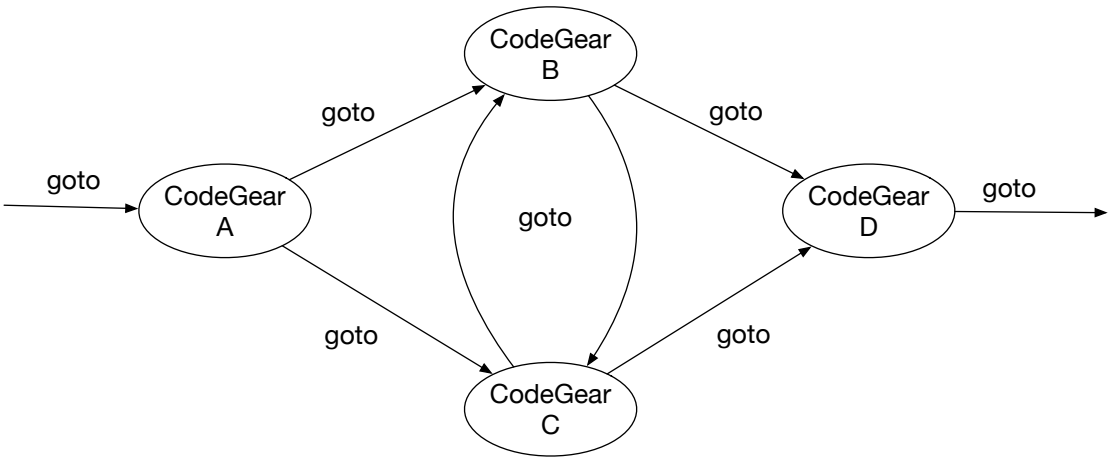

図 2.1: CodeGear 間の継続

現在 CbC は C コンパイラである GCC[2] 及び LLVM[3] 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.

2.1　CodeGear

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

ソースコード 2.1:　CodeGear の継続の例

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

```
7 __code cg1(Integer c){
8   goto cg2(c);
9 }
```

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

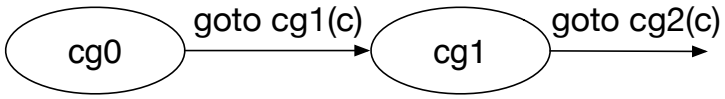

図 2.2: ソースコード 2.1 が表している CodeGear の状態遷移

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

2.2 DataGear

第3章　GearsOS

3.1　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.

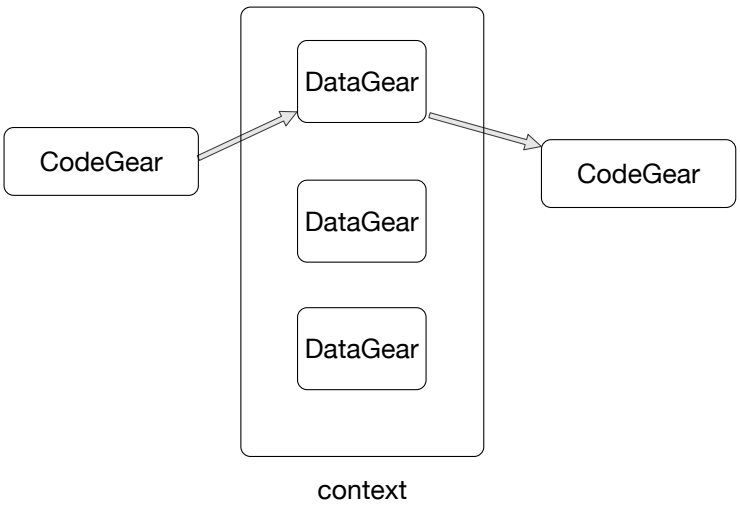

図 3.1: CodeGear,DataGear,contxt の関係図

3.2　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 先行研究 [4] において作成された Stack の Interface とその実装を表したものである．

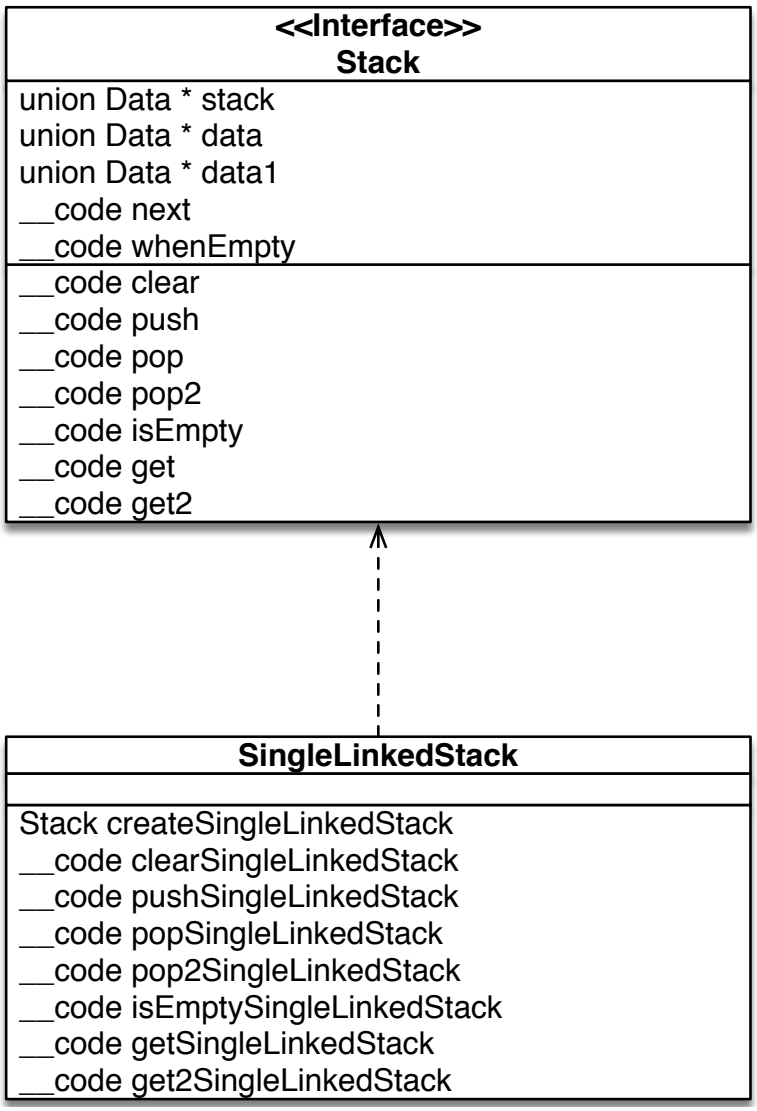

図 3.2: Stack の Interface とその実装

第4章　xv6

xv6 [5] 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．

4.1　xv6 の FileSystem 構造

xv6 の FileSystem の構造を以下の図 4.1 に示す．

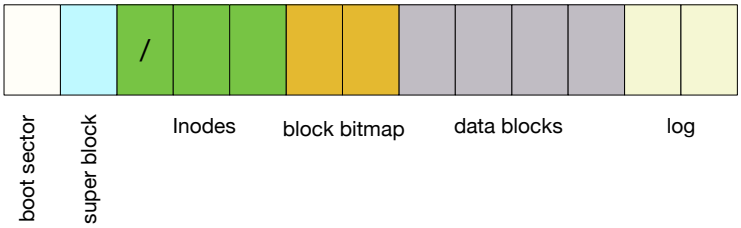

図 4.1: xv6 の FileSystem Layout

4.2

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

```
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,
           __code next(int ret, ...));
35     __code writei(Impl* fs, struct inode* ip, char* src, uint off, uint tot, uint n,
           __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;
```

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] 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.

[4] 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.

[5] Xv6, a simple Unix-like teaching operating system. `https://pdos.csail.mit.edu/6.828/2019/xv6.html`. (2019 年 10 月 19 日閲覧).

[6] 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