xv6の構成要素の継続の分析

清水 隆博[1,a)]　河野 真治[2,b)]

概要：OS 自体そのものは高い信頼性が求められるが、 OS を構成するすべての処理をテストするのは困難である。テストを利用して信頼性を高めるのではなく、 OS の状態を状態遷移を基本としたモデルに変換し形式手法を用いて信頼性を高めたい。
状態遷移単位での記述に適した言語である CbC を用いて、小さな unix である xv6 kernel の書き換えを行っている。このためには現状の xv6 kernel の処理がどのような状態遷移を行うのかを分析し、継続ベースでのプログラミングに変換していく必要がある。本稿では xv6kernel の構成要素の一部に着目し、状態遷移系の分析と状態遷移系を元に継続ベースで xv6 の再実装を行う。

1. OS の信頼性

様々なアプリケーションは OS の上で動作するのが当たり前になってきた。アプリケーションの信頼性を向上させるのはもとより、 土台となる OS 自体の信頼性は高く保証されていなければならない。OS そのものも巨大なプログラムであるため、 テストコードを用いた方法で信頼性を確保する事が可能である。しかし並列並行処理などに起因するバグや、 そもそも OS を構成する処理が巨大であることから、 テストで完全にバグを発見するのは困難である。テスト以外の方法で OS の信頼性を高めたい。

数学的な背景に基づく形式手法を用いて OS の信頼性を向上させることを検討する。OS を構成する要素をモデル検査してデッドロックなどを検知する方法や、 定理証明支援系を利用した証明ベースでの信頼性の確保などの手法が考えられる。形式手法で信頼性を確保するには、 まず OS の処理を証明などがしやすい形に変換して実装し直す必要がある。これに適した形として、 状態遷移モデルが挙げられる。OS の内部処理の状態を明確にし、 状態遷移モデルに落とし込むことでモデル検査などを通して信頼性を向上させたい。既存の OS はそのままに処理を状態遷移モデルに落とし込む為には、 まず既存の OS の処理中の状態遷移を分析し、仕様記述言語などによる再実装が必要となる。しかし仕様記述言語や定理証明支援系では、 実際に動く OS と検証用の実装が別の物となってしまうために、 C 言語などでの実装の段階で発生するバグを取り除くことができない。実装のソースコードと検証用のソースコードは近いセマンティクスでプログラミングする必要がある。

さらに本来行いたい処理の他に、メモリ管理やスレッド、CPU などの資源管理も行う必要がある。本来計算機で実行したい計算に必要な計算をメタ計算と呼び、 意図して行いたい処理をノーマルレベルの計算と呼ぶ。ノーマルレベル上での問題点をメタ計算上で発見し信頼性を向上させたい。プログラマからはノーマルレベルの計算のみ実装するが、整合性の確認や拡張を行う際にノーマルレベルと同様の記述力でメタ計算も実装できる必要がある。

ノーマルレベルの計算とメタ計算の両方の実装に適した言語として Continuation Based C(CbC) がある。CbC は C と互換性のある C の下位言語であり、 状態遷移をベースとした記述に適したプログラミング言語である。C との互換性のために、 CbC のプログラムをコンパイルすることで動作可能なバイナリに変換が可能である。また CbC の基本文法は簡潔であるため、 Agda などの定理証明支援系との相互変換や、 CbC 自体でのモデル検査が可能であると考えられる。すなわち CbC を用いて状態遷移を基本とした単位でプログラミングをすると、 形式手法で証明が可能かつ実際に動作するコードを記述できる。

現在小さな unix である xv6 kernel を CbC を用いて再実装している。再実装の為には、 既存の xv6 kernel の処理の状態遷移を分析し、継続を用いたプログラムに変換していく必要がある。本論文ではこの書き換えに伴って得られた xv6 kernel の継続を分析し、 現在の CbC による書き換えについて述べる。

[1] 琉球大学大学院理工学研究科情報工学専攻
[2] 琉球大学工学部工学科知能情報コース
[a)] anatofuz@cr.ie.u-ryukyu.ac.jp
[b)] kono@ie.u-ryukyu.ac.jp

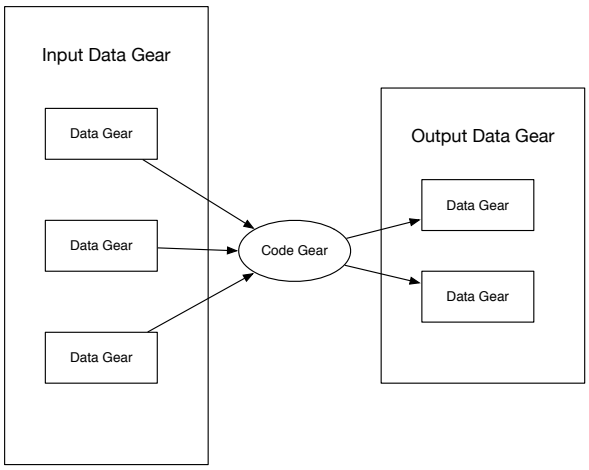

図 1: CodeGear と入出力の関係図

2. Continuation Based C

Continuation Based C(CbC) とは C 言語の下位言語であり、関数呼び出しではなく継続を導入したプログラミング言語である。CbC では通常の関数呼び出しの他に、関数呼び出し時のスタックの操作を行わず、次のコードブロックに `jmp` 命令で移動する継続が導入されている。この継続は Scheme などの環境を持つ継続とは異なり、スタックを持たず環境を保存しない継続である為に軽量である事から軽量継続と呼べる。また CbC ではこの軽量継続を用いた再帰呼び出しを利用することで `for` 文などのループ文を廃し、関数型プログラミングに近いスタイルでプログラミングが可能となる。現在 CbC は GCC 及び LLVM/clang 上にそれぞれ実装されている。

CbC では関数の代わりに CodeGear という単位でプログラミングを行う。CodeGear は通常の C の関数宣言の返り値の型の代わりに `__code` で宣言を行う。各 CodeGear は DataGear と呼ばれるデータの単位で入力を受け取り、その結果を別の DataGear に書き込む。入力の DataGear を InputDataGear と呼び、出力の DataGear を OutputDataGear と呼ぶ。CodeGear がアクセスできる DataGear は、InputDataGear と OutputDataGear に限定される。これらの関係図を図 1 に示す。

CbC で階乗を求める例題を Code 1 に示す。例題では CodeGear として `factorial` を宣言している。`factorial` は CodeGear の引数として `struct F` 型の変数 `arg` を受け取り、`arg` のメンバー変数によって `factorial` の再帰呼び出しを行う。CodeGear の呼び出しは `goto` 文によって行われる。この例題を状態遷移図にしたものを図 2 に示す。図中の四角が DataGear、円が CodeGear に対応する。

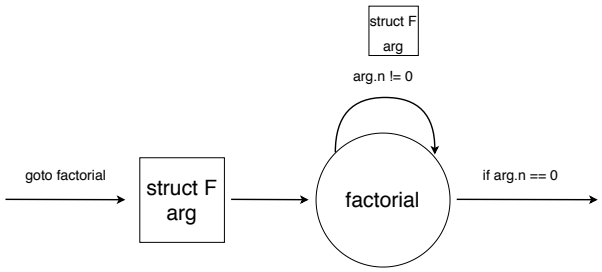

図 2: CbC で階乗を求める例題の状態遷移

```
__code factorial(struct F arg) {
    if (arg.n<0) {
        exit(1);
    }
    if (arg.n==0) {
      goto arg.next(arg);
    } else {
      arg.r *= arg.n;
      arg.n--;
      goto factorial(arg);
    }
}
```

Code 1: CbC で階乗を求める例題

CodeGear は関数呼び出し時のスタックを持たない為、一度ある CodeGear に遷移してしまうと元の処理に戻ってくることができない。しかし CodeGear を呼び出す直前のスタックは保存されるため、部分的に CbC を適用する場合は CodeGear を呼び出す `void` 型などの関数を経由することで呼び出しが可能となる。

この他に CbC から C へ復帰する為の API として、環境付き goto という機能がある。これは GCC では内部コードを生成、LLVM/clang では `setjmp` と `longjmp` を使うことで CodeGear の次の継続対象として呼び出し元の関数を設定することが可能となる。したがってプログラマから見ると、通常の C の関数呼び出しの返り値を CodeGear から取得する事が可能となる。

3. CbC を用いた OS の実装

軽量継続を持つ CbC を利用して、証明可能な OS を実装したい。その為には証明に使用される定理証明支援系や、モデル検査機での表現に適した状態遷移単位での記述が求められる。CbC で使用する CodeGear は、状態遷移モデルにおける状態そのものとして捉えることが可能である。CodeGear を元にプログラミングをするにつれて、CodeGear の入出力の Data も重要であることが解ってきた。CodeGear とその入出力である DataGear を基本とした OS として、GearsOS の設計を行っている。現在の GearsOS は並列フレームワークとして実装されており、実用的な OS のプロトタイプ実装として既存の OS 上への実装を目指している。

GearsOS では、CodeGear と DataGear を元にプログラミングを行う。遷移する各 CodeGear の実行に必要なデータの整合性の確認などのメタ計算は、MetaCodeGear と

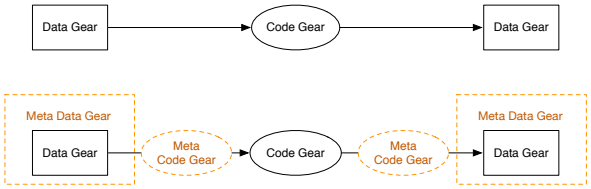

図 3: CodeGear と MetaCodeGear

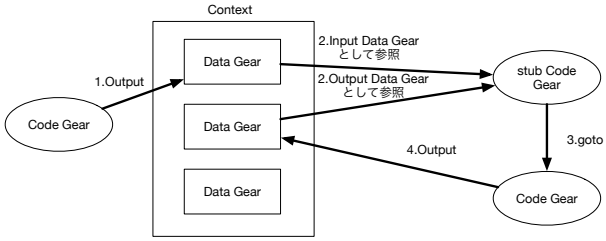

図 4: Context と各データの関係図

呼ばれる各 CodeGear ごと実装された CodeGear で計算を行う。この MetaCodeGear の中で参照される DataGear を MetaDataGear と呼ぶ。また、 対象の CodeGear の直前で実行される MetaCodeGear を StubCodeGear と呼ぶ。MetaCodeGear や MetaDataGear は、プログラマが直接実装することはなく、 現在は Perl スクリプトによって GearsOS のビルド時に生成される。CodeGear から別の CodeGear に遷移する際の DataGear などの関係性を、図 3 に示す。

通常のコード中では入力の DataGear を受け取り CodeGear を実行、 結果を DataGear に書き込んだ上で別の CodeGear に継続する様に見える。この流れを図 3 の上段に示す。しかし実際は CodeGear の実行の前後に実行される MetaCodeGear や入出力の DataGear を保存場所から取り出す MetaDataGear などのメタ計算が加わる。これは図 3 の下段に対応する。

遷移先の CodeGear と MetaCodeGear の紐付けや、 計算に必要な DataGear を保存や管理を行う MetaDataGear として context がある。context は処理に必要な CodeGear の番号と MetaCodeGear の対応表や、DataGear の格納場所を持つ。計算に必要なデータ構造と処理を持つデータ構造であることから、 context は従来の OS のプロセスに相当するものと言える。cotnext と各データ構造の関わりを図 4 に示す。

コード上では別の CodeGear に直接遷移している様に見えるが、 実際は context 内の遷移先の CodeGear に対応するスロットから、対応する MetaCodeGear に遷移する。MetaCodeGear 中で、次に実行する CodeGear で必要な DataGear を context から取り出し、 実際の計算が行われる。

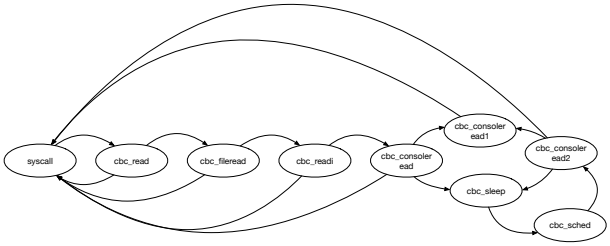

図 5: read システムコールの状態遷移

4. xv6 kernel

xv6 とはマサチューセッツ工科大学で v6 OS を元に開発された教育用の UNIX OS である。xv6 は ANSI C で実装されており、x86 アーキテクチャ上で動作する。Raspberry Pi 上での動作を目的とした ARM アーキテクチャのバージョンも存在する。本論文では最終的に Raspberry Pi 上での動作を目指しているために、 ARM アーキテクチャ上で動作する xv6 を扱う。

xv6 は小規模な OS だがファイルシステム、 プロセス、システムコールなどの UNIX の基本的な機能を持つ。またユーザー空間とカーネル空間が分離されており、 シェルや ls などのユーザーコマンドも存在する。

本論文では xv6 のファイルシステム関連の内部処理と、システムコール実行時に実行される処理について分析を行う。xv6 kernel のファイルシステムは階層構造で表現されており、 最も低レベルなものにディスク階層、 抽象度が最も高いレベルのものにファイル記述子がある。

本論文では xv6 の継続の分析をシステムコール部分とファイルシステム、 仮想メモリなどの OS の根幹部分でそれぞれ行った。

5. xv6 のシステムコールの継続の分析

xv6 の処理を継続を中心とした記述で再実装を行う。この際に、 xv6 のどの処理に着目するかによって継続の実装が異なっていくことが実装につれてわかった。

まず xv6 の `read` システムコールに着目し、 システムコール内部でどのような状態を遷移するかを分析した。分析結果を CbC の CodeGear に変換し、 状態遷移図におこしたものを図 5 に示す。

CbC で再実装した `read` システムコールは、 xv6 の `read` システムコールのディスパッチ部分から、`cbc_read`CodeGear に `goto` 文で軽量継続される。継続後は read する対象によって `cbc_readi` や、 `cbc_consoleread` などに状態が変化していく。各 CodeGear の遷移時には DataGear がやり取りされる。DataGear は xv6 のプロセス構造体に埋め込まれた context を経由して CodeGear に渡される。

この実装の利点として、 CodeGear の命名と状態が対応

しており、 状態遷移図などに落としても自然言語で説明が可能となる点が挙げられる。しかし実際には cbc_readi の状態はさらに複数の CodeGear に分離しており、 実際に read システムコールを実装する CodeGear の数は図の状態より多い。この事から、 複数の CodeGear を 1 つにまとめた上で見た状態と、 各 CodeGear それぞれの状態の 2 種類の状態があるといえる。

複数の CodeGear をまとめた状態は、 抽象化した API の操作時におけるアルゴリズム上の問題が無いかの確認として使用出来る。対して各 CodeGear それぞれはモデル検査や、 特定の関数の中の処理が適しているかどうかの検査として見ることが出来ると考えられる。

この事から GearsOS では、 各 CodeGear のモジュール化の仕組みである Interface 機能を導入している。Interface の導入によって CodeGear を定義することで状態数を増やしても、 抽象化された API を利用することで細部の状態まで意識する必要が無くなった。xv6 の処理を CbC で再実装する際には、 対象の継続の API をまず決定しモジュール化を図る必要がある。

6. xv6 のシステムコール以外の継続の分析

xv6 はシステムコール以外に、 ファイルシステムの操作やページテーブルの管理などの処理も存在している。これらは OS の立ち上げ時やシステムコールの中で、ファイルシステムの操作に対応した関数や構造体などの API を通して操作される。システムコールの一連の流れに着目するのではなく、 特定の対象の API に着目して継続の分析を検討した。

xv6 のファイルシステムに関する関数などの API は主に fs.c 中に記述されている。Code2 に示す様に、 fs.c 中に定義されている API を抜き出し、CbC の Interface として定義した。__code から始まる CodeGear の名前が、 それぞれ抽象化された CodeGear の集合の最初の継続となる。

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

Code 2: ファイルシステム操作の API の一部

Code2 内の readsb などは fs.c 内で定義されている C の関数名と対応している。この C の関数を更に継続ごと分割するために、 関数内の if 文などの分岐を持たない基本単位である Basic Block に着目した。

CbC の CodeGear の粒度は C の関数とアセンブラの中間であるといえるので、 BasicBlock を CodeGear に置き換える事が可能である。したがって特定の関数内の処理の BasicBlock を分析し、 BasicBlock に対応した CodeGear へ変換することが可能となる。実際に BasicBlock 単位で切り分ける前の処理と、切り分けたあとの処理の一部を示す。例として inode のアロケーションを行う API でる ialloc の元のコードを Code3 に示す。

```
struct inode* ialloc (uint dev, short type)
{
    readsb(dev, &sb);
    for (inum = 1; inum < sb.ninodes; inum++) {
        bp = bread(dev, IBLOCK(inum));
        dip = (struct dinode*) bp->data + inum % IPB;

        if (dip->type == 0) { // a free inode
            memset(dip, 0, sizeof(*dip));
            // omission
            return iget(dev, inum);
        }
        brelse(bp);
    }
    panic("ialloc:␣no␣inodes");
}
```

Code 3: ialloc の元のソースコード

ialloc はループ条件である inum < sb.ninodes が成立しなかった場合は panic へと状態が遷移する。この for 文での状態遷移を CodeGear に変換したものを Code4 に示す。

```
__code allocinode_loopcheck(struct fs_impl* fs_impl,
     uint inum, uint dev, struct superblock* sb,
     struct buf* bp, struct dinode* dip, __code next
     (...)){
    if( inum < sb->ninodes){
        goto allocinode_loop(fs_impl, inum, dev, type,
              sb, bp, dip, next(...));
    }
    char* msg = "failed␣allocinode...";
    struct Err* err = createKernelError(&proc->
         cbc_context);
    goto err->panic(msg);
}
```

Code 4: ループ条件を確認する CodeGear

Code4 ではループ条件の成立を if 文で確認し、ループ処理に移行する場合は allocinode_loop へ遷移する。ループ条件が満たされなかった場合は、 コンテキストから

panic に関する CodeGear の集合を取り出し、 集合中の panic CodeGear へと遷移する。オリジナルの処理では、ループ中に dip->type == 0 が満たされた場合は関数から return 文により関数から復帰する。CodeGear では Code5 内で、 状態が分けられる。この先の継続は、 復帰用の CodeGear かループの先頭である allocinode_loopcheck に再帰的に遷移するかになる。

```
__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
     (...)){
    bp = bread(dev, IBLOCK(inum));
    dip = (struct dinode*) bp->data + inum % IPB;
    if(dip->type = 0){
        goto allocinode_loopexit(fs_impl, inum, dev,
             sb, bp, dip, next(...));
    }

    brelse(bp);
    inum++;
    goto allocinode_loopcheck(fs_impl, inum, dev, type
         , sb, bp, dip, next(...));
}
```

Code 5: ループ中に復帰するかどうかの確認をする CodeGear

この継続の分析方法の利点として、 既存のコードの Basic Block 単位で CodeGear に変換可能であるため機械的に CodeGear への変更が可能となる。既存の関数上のアルゴリズムや処理に殆ど変更がなく変更可能であるために、 CodeGear で細分化して表現することは容易となる。

現在は従来の xv6 の関数呼び出しを元にした API の中で CodeGear に分割している。このために既存の API 内の処理の細分化は可能とはなったが、 API そのものを CodeGear を用いた継続に適した形には表現できていない。API の内部の CodeGear はあくまで Basic Block 単位に基づいているために、状態遷移図で表現した際に自然言語で表現できない CodeGear も存在してしまう。

さらに、 for ループを CodeGear に分割することを考えるとループ中にループの index を利用している場合は、その index も次の継続に渡さなければならない。このため index を使用していない CodeGear でも継続の引数として index を受け取り、 実際に index を利用する CodeGear に伝搬させる必要がある。これらの問題を解決する為には、 API を分割した CodeGear それぞれの DataGear に型を与え、 どの継続で DataGear の意味が変わるかを追求する必要がある。API を分割して作成した CodeGear の DataGear は、現在各 API に対応した 1 つの巨大な構造体に隠蔽されている。巨大な構造体で管理するのではなく、構造体で次の CodeGear の状態に影響を与える要素を適宜組み合わせた DataGear を作る必要がある。

7. CbC を用いた部分的な xv6 の書き換え

CbC では CodeGear、 DataGear からなる単位を基本とし、 それぞれにメタな Gear が付随する。また実行に必要な CodeGear と DataGear をまとめた context という MetaDataGear が存在する。この機能を元に xv6 の書き換えを検討した。

xv6 内で CbC の軽量継続に突入する際は、 元の処理関数に通常の方法では戻ってくることができず、部分的に書き換えていくのが困難である。今回は呼び出し関数に戻れるスタックフレームを操作したい為に、 ダミーの void 関数を用意した。この関数内で CodeGear に goto 文を用いて遷移することで、 CbC から帯域脱出した際に void 関数の呼び出し元から処理を継続し、部分的に CbC に書き換えることが可能となった。Code6 では、 userinit 関数へ戻るために、 cbc_init_vmm_dumy を経由している。

```
void cbc_init_vmm_dummy(struct Context* cbc_context,
     struct proc* p, pde_t* pgdir, char* init, uint sz
     )
{
    struct vm* vm = createvm_impl(cbc_context);
    goto vm->init_vmm(vm, pgdir, init, sz , vm->
         void_ret);
}

void userinit(void)
{
// omission

    if((p->pgdir = kpt_alloc()) == NULL) {
        panic("userinit:␣out␣of␣memory?");
    }

    cbc_init_vmm_dummy(&p->cbc_context, p, p->pgdir,
         _binary_initcode_start, (int)
         _binary_initcode_size);

    p->sz = PTE_SZ;
    memset(p->tf, 0, sizeof(*p->tf));
// omission
}
```

Code 6: 部分的に CbC を適応する例

Code6 中で、 CodeGear への遷移が行われる goto vm->init_vmm() の vm->void_ret は init_vmm の次の継続の CodeGear 名である。この vm->void_ret は return するのみの CodeGear であり、 void 型関数と組み合わせることで呼び出し元へと復帰する事が可能となる。

8. xv6 の今後の再実装

xv6 ではカーネルパニックの発生時や、 inode のキャッシュなどをグローバル変数として利用している。グローバル変数を使用してしまうと、 CodeGear で定義した状態が

DataGear 以外のグローバル変数によって変更されてしまう。グローバル変数を極力使わず継続を中心とした実装を行いたい。

context は現在プロセス構造体に埋め込まれており、kernel そのものの状態を制御するためには各 context を管理する機能が必要であると考えられる。context を管理する方法として、 各 context 間でメッセージなどをやりとりする方法や、 cotnext を主軸にして割り込みなどの処理を再検討するものがある。他には kernel そのものの context を定義し、 kernel 全体の状態とプロセスの状態をそれぞれ別の cotnext で処理をするというのも検討している。

現状は xv6 の全ての機能をまだ CbC を用いて再実装をしていない。ファイルシステムや仮想メモリにまつわる処理などは API 単位では再実装しているが、 API を呼び出す箇所は C の関数上で部分的に呼び出している。そのために OS そのものを状態遷移単位で完全に書き直す必要が存在し、 そのためには全ての処理に対して状態を定義していく必要がある。xv6 にはアセンブラで記述されている処理も複数存在するため、 CodeGrar の表現においてこのような処理の扱いも決定する必要がある。

また OS 上で DataGear と CodeGear の位置づけを明確に定義する必要も存在する。現在は関数を分割した状態として CodeGear を定義している。DataGear の依存関係や CodeGear の並列実行など、 プロセスベースで実装していた処理を CodeGear などで意味がある形式にする必要がある。

また xv6 にはユーザーコマンドも存在しているために、ユーザーコマンド向けの CbC の API なども考慮したい。

9. まとめ

本稿では xv6 を継続を用いた単位での再実装を検討し、実際にいくつかの処理を再実装した。現状はまだ xv6 の実装を利用した証明は行っていない。今後はモデル検査などを行い、 OS の信頼性を向上させていきたい。それに伴って、 xv6 自体にモデル検査機能の組み込みなども行っていく。また Agda などの定理証明支援系で証明された処理から、 CbC の CodeGear に変換する処理系の実装なども検討する。

参考文献

[1] Andrew S.Tanenbaum, H. B.: Modern Operating Systems (2015).
[2] : Raspberry Pi — Teach, Learn, and Make with Raspberry Pi, https://www.raspberrypi.org.
[3] Wang, Z.: xv6-rpi, https://code.google.com/archive/p/xv6-rpi/ (2013).
[4] TOKKMORI, K. and KONO, S.: Implementing Continuation based language in LLVM and Clang, *LOLA 2015* (2015).
[5] Moggi, E.: Notions of Computation and Monads, *Inf. Comput.*, Vol. 93, No. 1, pp. 55–92 (online), DOI: 10.1016/0890-5401(91)90052-4 (1991).
[6] Yang, J. and Hawblitzel, C.: Safe to the Last Instruction: Automated Verification of a Type-safe Operating System, *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, New York, NY, USA, ACM, pp. 99–110 (online), DOI: 10.1145/1806596.1806610 (2010).
[7] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H. and Winwood, S.: seL4: Formal Verification of an OS Kernel, *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, New York, NY, USA, ACM, pp. 207–220 (online), DOI: 10.1145/1629575.1629596 (2009).
[8] Sigurbjarnarson, H., Bornholt, J., Torlak, E. and Wang, X.: Push-button Verification of File Systems via Crash Refinement, *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, Berkeley, CA, USA, USENIX Association, pp. 1–16 (online), available from ⟨http://dl.acm.org/citation.cfm?id=3026877.3026879⟩ (2016).
[9] Nelson, L., Sigurbjarnarson, H., Zhang, K., Johnson, D., Bornholt, J., Torlak, E. and Wang, X.: Hyperkernel: Push-Button Verification of an OS Kernel, *Proceedings of the 26th Symposium on Operating Systems Principles* (2017).
[10] MASATAKA, H. and KONO, S.: GearsOS の Hoare Logic をベースにした検証手法，ソフトウェアサイエンス研究会 (2019).
[11] Norell, U.: Dependently Typed Programming in Agda, *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, New York, NY, USA, ACM, pp. 1–2 (online), DOI: 10.1145/1481861.1481862 (2009).
[12] Lattner, C. and Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California (2004).
[13] GNU Compiler Collection (GCC) Internals: *http://gcc.gnu.org/onlinedocs/gccint/*.
[14] 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， No. 4 (2019).
[15] 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) (2016).
[16] ARM Architecture Reference Manual: *http://infocenter.arm.com/help/topic/com.arm.doc.subset.architecture.reference/index.html*.
[17] Russ Cox, Frans Kaashoek, Robert Morris: xv6 a simple, Unix-like teaching operating system, https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf.
[18] Chen, H., Ziegler, D., Chajed, T., Chlipala, A., Kaashoek, M. F. and Zeldovich, N.: Using Crash Hoare Logic for Certifying the FSCQ File System, *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, New York, NY, USA, ACM, pp. 18–37 (online), DOI: 10.1145/2815400.2815402 (2015).