修士 (工学) 学位論文

Master's Thesis of Engineering

CbC インターフェースによる CbCXv6 の書き換え

2020 年 3 月

March 2020

桃原 優

Yu Tobaru

琉球大学

大学院理工学研究科

情報工学専攻

Information Engineering Course

Graduate School of Engineering and Science

University of the Ryukyus

指導教員：教授 玉城 史朗

Supervisor: Prof. Shiro Tamaki

本論文は、修士 (工学) の学位論文として適切であると認める。

論　文　審　査　会

印

(主　査)　　玉城 史朗

印

(副　査)　　遠藤 聡志

印

(副　査)　　名嘉村 盛和

印

(副　査)　　河野　真治

要旨

OS 自体に信頼性が求められるが、OS の全てのコードに対して検証を行うのは困難である。

本研究室で信頼性を保障するのに適したプログラミング言語 CbC を開発してきた。CbC を使って OS の開発をすることにより、CbC の軽量継続により OS の動作を状態遷移を基本としたモデルに落とし込むことができる。これにより状態遷移機械の検証手法を OS に対して適用し信頼性を高めることができるようになると考えられる。本研究では小型の OS である xv6 を CbC で書き換えることの一部を行う。具体的には OS のメモリ管理を担当する vm.c を CbC に変換する。

CbC ではインターフェースを用いたモジュール化を行う。vm.cbc はメモリ管理 (特に paging) を行うインターフェースと、その実装として記述される。インターフェース内部での CbC の接続はメタレベルの記述になる。OS で使われるすべてのコードとデータは context と呼ばれるメタデータに記述される。context のデータを書き換えることにより、xv6 で仮想 OS やコンテナを定義できるようになると考えられる。

本論文では OS の信頼性の基本であるメモリ管理部分のインターフェースと実装の記述の考察を行う。

Abstract

Reliability of Operating Systems are important, but it is difficult to verify the source code of the Operating System.

Programming language CbC is developed in our laboratory, which is designed to support reliability. Operating System Project has automaton like descriptions if it is written in CbC. Using automaton based verification technologies, the Operating System becomes more dependable. In this research, we will rewrite x.v6 kernel which is a small operating system written in C. Actually, we will rewrite the memory management part vm.c of it using CbC.

CbC has module system called interfaces. vm.cbc is implemented as an interface and the implementations. In the interfaces, all connections and data structures are written a meta data, which is called Context. Virtual OS or so called Container can be realized as a modification of the Context.

In this paper, we describes rewriting details of memory management interface.

目 次

図 目 次

表 目 次

第1章　OS の信頼性の保障

OS を要する機器に依存している現代では、OS のバグは日常に支障を来たすことに繋がる。実際にパスワードなしで root にアクセスできてしまう (https://support.apple.com/ja-jp/HT208331) など、クリティカルなバグも発生した。

OS 自体に信頼性が求められるが、複雑な機能が多く、全てのコードに対して検証を行うのは困難である。CPU やメモリなどの資源管理は基本的には OS が行なっている。これは資源管理が複雑な上、アクセスされたり書き換えられることを防ぐためだと考えられる。

このように OS には資源管理やカーネルの処理などの部分が存在し、メタレベルの計算と呼ぶ。それ以外の処理をノーマルレベルの計算と呼ぶ。

本研究室ではノーマルレベルとメタレベルの記述を行える CbC[3] というプログラミング言語を開発してきた。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、その間を関数型プログラミング言語のように goto によって継続する。そのため、状態遷移を基本に記述することができる。Code Gear に対して入力の Data Gear と出力の 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 を使って 信頼性の保証と拡張性を持たせる Gears OS[5] の開発を行なっている。本論文では、xv6[4] という OS を参考にした Geas OS の書き換えの説明を行う。OS の信頼性の基本であるメモリ管理部分を書き換えることで信頼性を保証したい。具体的には Page のバリデーションチェックによる不正なデータの変更やサンドボックスによるエクセプションを飛ばすなどが挙げられる。また、Gears OS のメタレベルとノーマルレベルでは書き換えなどを防ぐために見えるデータに違いが生じ、Code Gear と Meta Code Gear の記述も煩雑になる。それを解消するために、インターフェースによるモジュール化を導入 [1] した。インターフェース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ことを目的する。

また、CbC は状態遷移ベースで記述される上に、実行される環境やデータも Stack を使わず遷移されていく。そのため、OS の書き換えを行った後に実行環境を複数用意し、それぞれで実行することで OS 内でコンテナや VM を実装できると考えられる。

第2章　CbC による Geas OS の開発

信頼性の保証と並列実行のサポートを目的として、 本研究室では CbC というプログラミング言語を開発してきた。LLVM[9] 上で実装された CbC と GCC[10] 上で実装された CbC が存在する。さらにその CbC を使って Gears OS を開発している。従来の OS が行うメモリ管理並列実行は Meta レベル (kernel space) で処理される。ノーマルレベルからメタレベルの記述ができる GearsOS を開発している。

2.1　Code Gear と Data Gear

Gears OS は Code Gear と Data Gear という単位でプログラムを記述する CbC を用いて実装する。Code Gear は CbC における最も基本的な処理の単位である。Code Gear 間で入力 (Input Data Gear) と出力 (Output Data Gear) を持ち、goto によって Code Gear から次の Code Gear へ遷移し、継続的に処理を行う。関数呼び出しとは異なり、呼び出し元には戻らない。Code Gear 間の処理の流れを図 2.1 に示す。

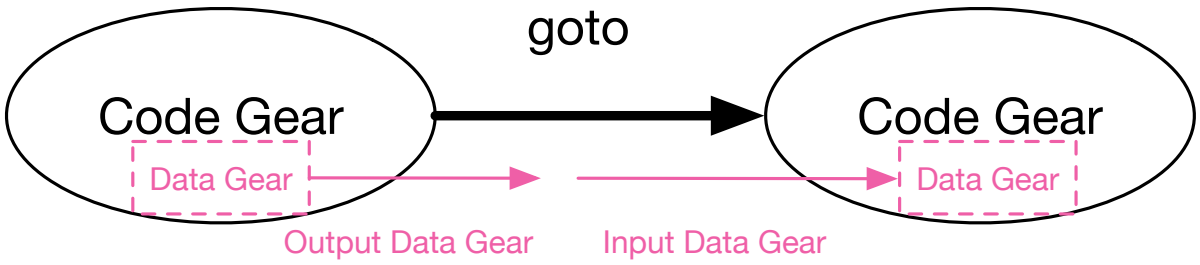

図 2.1: Code Gear 間の継続

Data Gear は CbC におけるデータの基本的な単位である。Input Data Gear と Output Data Gear があり、Code Gear の遷移の際に Input Data Gear を受け取り、Output Data Gear を書き出す。

2.2 Meta Code Gear と Meta Data Gear

CbC ではノーマルレベルの記述と別にメタレベルで記述することができる。メタレベルとは、メモリや CPU などの資源管理を行える部分で C 言語だと systemcall で呼び出し扱う処理で、Linix だとカーネル空間に相当する。メタレベルからノーマルレベルの記述の正しさを証明する。

メタ計算は Meta Code Gear と Meta Data Gear を用いる。この 2 つはノーマルレベルからメタレベルの変換する時に使われる。メタレベルの変換は Perl スクリプトによる cmake で実装している。Gears OS での Meta Code Gear は Code Gear の直前、 直後に挿入され、メタ計算を実行する。それぞれの Code Gear, Meta Code Gear の継続には入力される Data Gear(Input Data Gear) と出力される Data Gear(Output Data Gear) が存在する。Meta Code Gear は自動生成だけではなく記述することも可能である。そうすることでメタ計算を記述することができるようになったり、goto による継続先を変更することで Geas OS の機能を置き換えることができる。Code Gear 間の継続はノーマルレベルでは 図 2.1 のように見えるが、メタレベルでの Code Gear は図 2.2 の下のように継続を行っている。

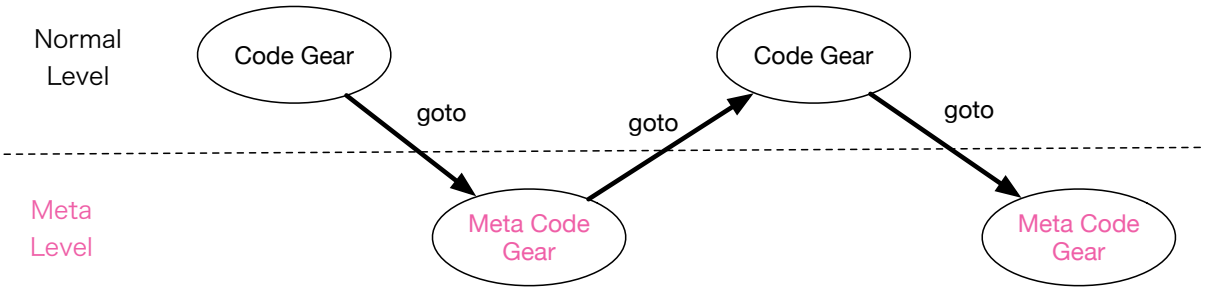

図 2.2: ノーマルレベルとメタレベルの継続の見え方

2.3 Context

Gears OS の Context は Meta Data Gear であり、接続可能な Code Gear と Data Gear のリスト、Data Gear を確保するメモリ空間などを持っている。Gears OS は 必要な Code Gear、 Data Gear を参照したい場合、Context を経由する必要がある。しかし、Context をノーマルレベルの計算から直接扱うと書き換えられるリスクが生じる。そこで Context から必要なデータを取り出して Code Gear に接続する Meta Code Gear を定義して、間接的に Data Gear にアクセスする。この Meta Code Gear を stub Code Gear と呼ぶ。

stub Code Gear は Code Gear ごとに生成され、生成元の Code Gear の直後に goto で継続される。

Context と stub Code Gear の関係を 図 2.3 に示す。Code Gear は Context が持つ Data Gear へのポインタを持っており、Context 内のその Data Gear にアクセスする。

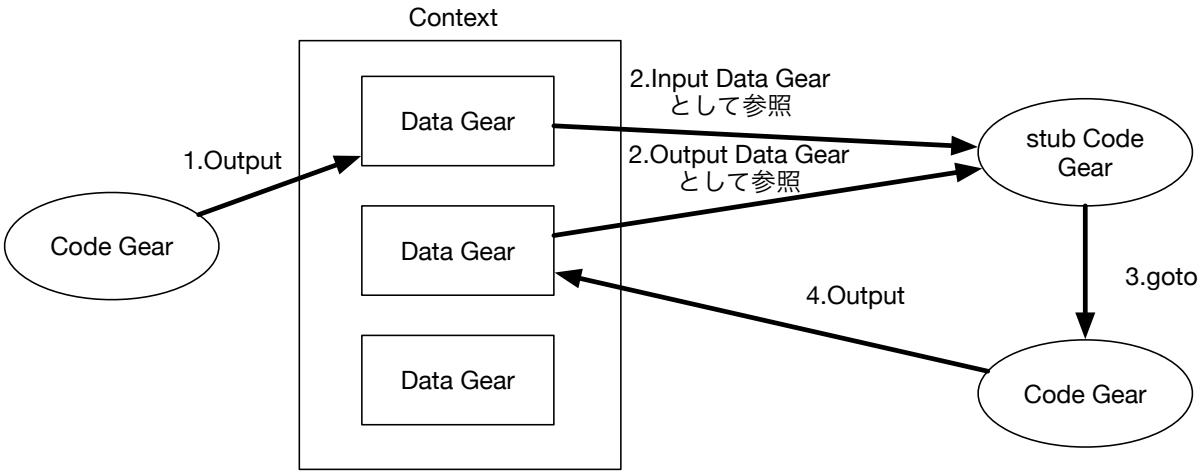

図 2.3: Context が持つ Data Gear へのアクセス [1]

第3章　xv6

xv6[4] とは、マサチューセッツ工科大の大学院生向け講義の教材として使うために、UNIX V6 という OS を ANSI-C(規格化された C 言語) に書き換え、x86 に移植した Xv6 OS である。

本研究では、この xv6 を参考に Gears OS の開発を行なっている。

3.1　Kernel Space と User Space

Xv6 は Kernel を採用している。 Kernel は OS にとって中核となるプログラムである。Xv6 では Kernel と User プログラムは分離されており、kernel はプログラムにプロセス管理、メモリ管理、I/O　やファイルの管理などのサービスを提供する。User プログラムは kernel に直接アクセスできない。これは重要なファイルを書き換えられたり、アクセスされるのを防ぐためだと考えられる。User プログラムが Kernel のサービスを呼び出す場合、system call を用いて User Space から Kernel Space Space へ入り実行される。Kernel は CPU のハードウェア保護機構を使用して、 User Space で実行されているプロセスが自身のメモリのみアクセスできるように保護している。User プログラムが system call をすると、ハードウェアが一時的に特権レベルを上げ、kernel のプログラムが実行される。この特権レベルを持つプロセッサの状態を kernel モード、特権のない状態を User モードと言う。

3.2　system call

User プログラムが Kernel の処理を行う場合、system call を用いる。User プログラムが system call を呼び出すと、トラップが発生する。トラップが発生すると、User プログラムは中断され、Kernel に切り替わり処理を行う。Xv6 の system call のリストを 3.1 に示す。

ソースコード 3.1: xv6 のシステムコールのリスト [1]

```
1 static int (*syscalls[])(void) = {
2         [SYS_fork]    =sys_fork,
```

```
3           [SYS_exit]    =sys_exit,
4           [SYS_wait]    =sys_wait,
5           [SYS_pipe]    =sys_pipe,
6           [SYS_read]    =sys_read,
7           [SYS_kill]    =sys_kill,
8           [SYS_exec]    =sys_exec,
9           [SYS_fstat]   =sys_fstat,
10          [SYS_chdir]   =sys_chdir,
11          [SYS_dup]     =sys_dup,
12          [SYS_getpid]  =sys_getpid,
13          [SYS_sbrk]    =sys_sbrk,
14          [SYS_sleep]   =sys_sleep,
15          [SYS_uptime]  =sys_uptime,
16          [SYS_open]    =sys_open,
17          [SYS_write]   =sys_write,
18          [SYS_mknod]   =sys_mknod,
19          [SYS_unlink]  =sys_unlink,
20          [SYS_link]    =sys_link,
21          [SYS_mkdir]   =sys_mkdir,
22          [SYS_close]   =sys_close,
23  };
```

3.3 Xv6-rpi

Xv6 は Arm のバイナリを出力するので、シングルボードコンピュータである Raspberry Pi や携帯電話など様々なハードウェアで動かすことができる。実際に Raspberry Pi 上で動かすために xv6-rpi[12] という OS を用意して動作を検証した。xv6-rpi は CbCXv6 と別で用意している。

xv6-rpi で USB 接続が可能か確認するためにシリアル通信を行った。Raspberry Pi のシリアルコンソール用の USB 変換ケーブルを使用し、機器は Raspberry Pi 3B+ を、接続先の OS は Ubuntu を使用した。Raspberry Pi 用のシリアルコンソールの USB 変換ケーブルを使用し、図 3.1 のように 6 番ピン (黒)、8 番 (白)、10 番 (緑) の順で接続する。赤のピンは繋げる必要はない。

シリアルコンソール用の USB 変換ケーブルを Raspberry Pi と Ubuntu に接続すると、dev ディレクトリ直下に ttyUSB0 として認識される。Ubuntu 側で 別のシステムを呼び出す cu コマンドを使い sudo cu -s 115200 -l /dev/ttyUSB0 と入力することでシリアル通信が可能となる。図 3.2 は Raspberry Pi と USB Serial で通信を行った際の画像である。Ubuntu 側で入力したコマンドが実行されていることが確認できた。

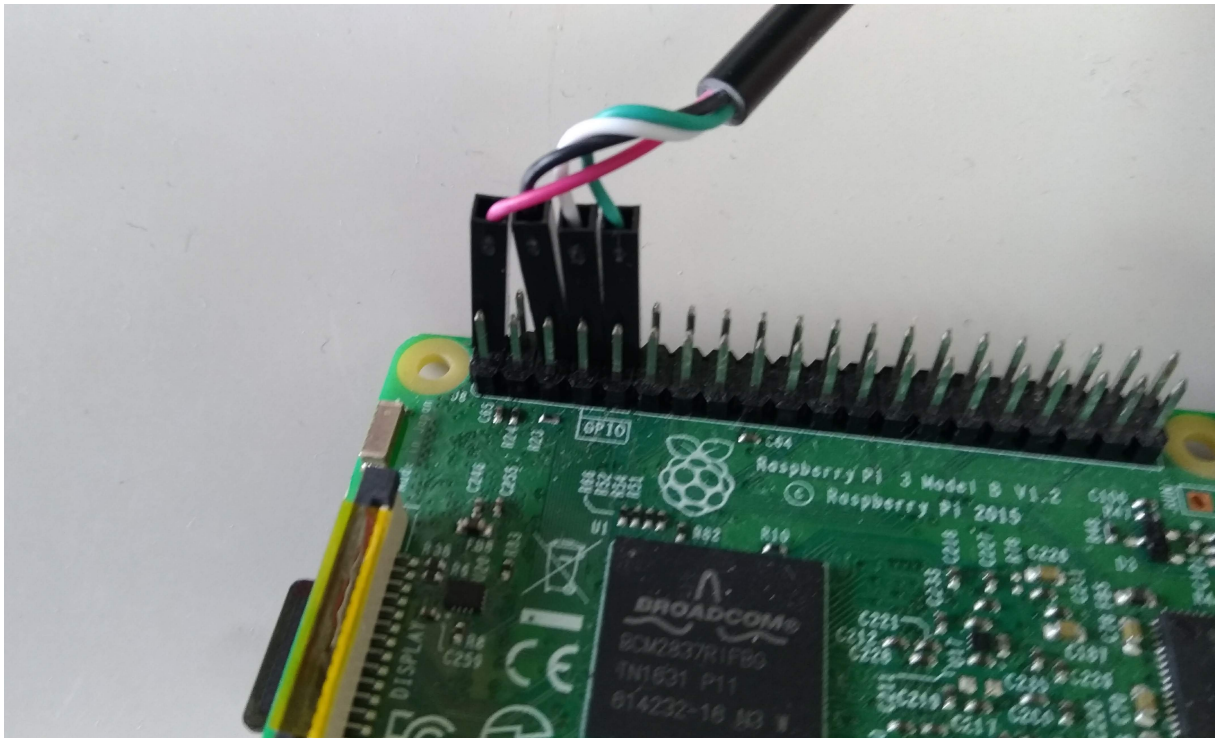

図 3.1: Raspberry Pi と シリアルコンソールの USB 変換ケーブルの接続

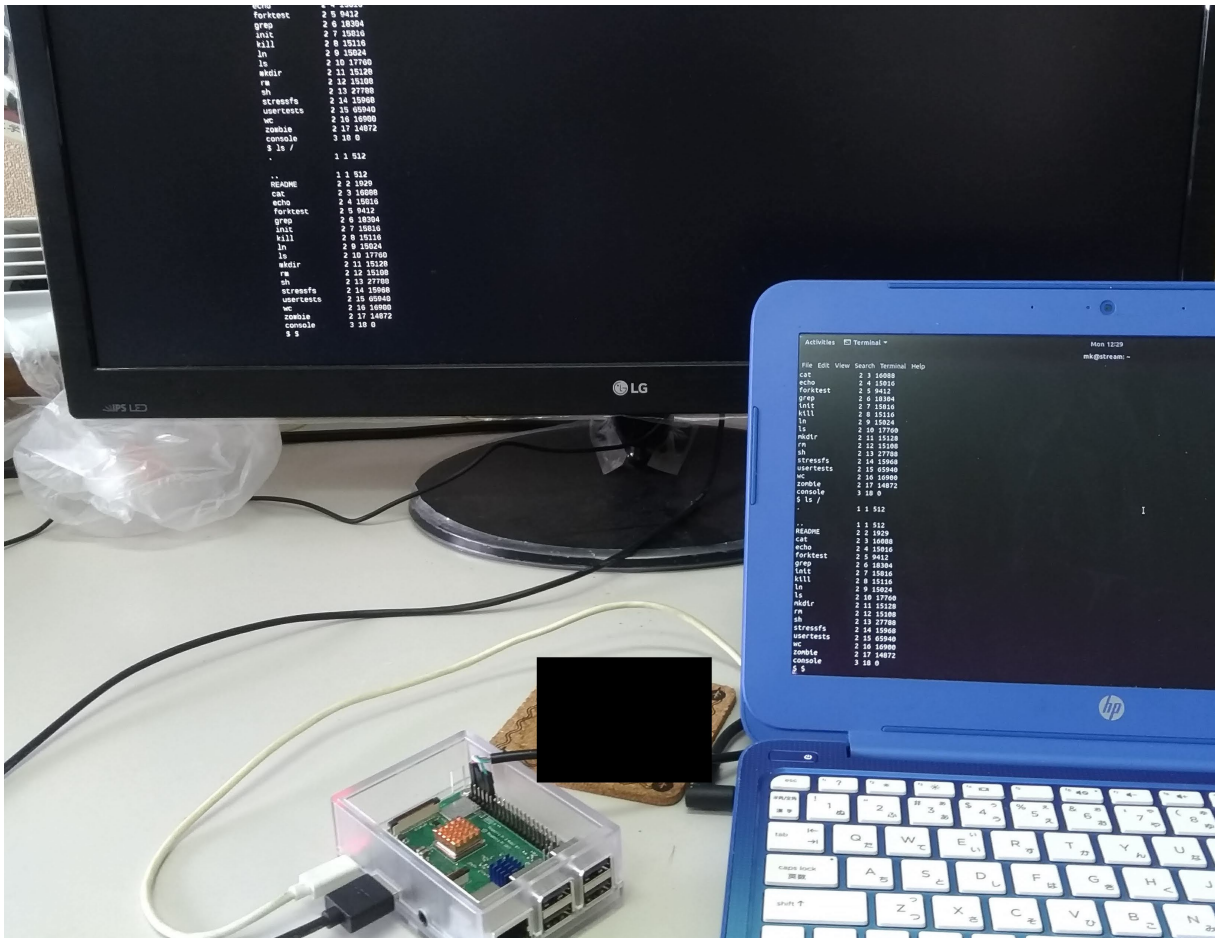

図 3.2: Raspberry Pi の USB シリアル通信

第4章　CbCXv6 のメモリ管理

OS の信頼性の基本である メモリ管理 の書き換えについて説明する。

4.1　Xv6 を元にした Gears OS の実装

Gears OS ではハードウェア上でメタレベルの計算や並列実行を行いたいので、Raspberry Pi でもバイナリを出力できる Xv6 を CbC で書き換える。ANSI-C で書かれている Xv6 を CbC に書き直し、それを元に Gears OS を実装していく。

4.2　Paging

実メモリをそのまま使うと様々な問題が生じる。ユーザープログラム側で空いているメモリ番地を探す必要がでてきたり、メモリ間にデータとして扱うには小さな隙間ができるフラグメンテーションが起こる。xv6 ではメモリ管理の手法の 1 つとして Paging を採用している。

Paging ではメモリを Page と呼ばれる固定長の単位に分割し、メモリとスワップ領域で Page を入れ替えて管理を行う。Paging を扱うことでブロック単位で管理することによりフラグメンテーションが解消でき、MMU が実メモリを管理することによってプログラム側で空いているメモリを探す必要がなくなる。

4.3　xv6 の Paging の主要関数

xv6 の Paging は vm.c で行われる。vm.c で使われる主要関数の役割について説明する。

init_vmm : 最初に外部ファイルの main.c から呼び出される関数。lock に使う側の id を入れ、誰も使ってなければ 0 を入れる。

kpt_free : カーネルのメモリを解放する。

kpt_alloc : メモリ割り当て。Page Table が NULL なら panic 関数を呼び出す。

walkpgdir : Page Table を探してアドレスを返す。loaduvm などの if 文内で呼び出される。

loaduvm : スワップ領域から呼び出され、カーネル内をループしながらユーザープロセスを待ち状態にする。

allocuvm : 仮想メモリをユーザープロセスに割り当てる。

inituvm : プロセス側である proc.c から呼び出され、Page ディレクトリを作成する。

4.4 CbCXv6 での Paging

Context に必要な Page Table を提供する Interface が必要である。Page Table に相当するデータを Input Data Gear で受け取って変更した後、Context にあるメモリコントロールを担当する Meta Data Gear に goto で遷移してアクセスする。メタレベルで処理することで カーネル側の処理である Page Table を操作することができる。Page Table Entry のバリデーションをチェックして反映することで、他のプロセスから Page Table を書き換えられることを防ぐ。また、サンドボックスにしておいて、他のプロセスが書き換えられた時にエクセプションを飛ばすようにすることで信頼性の保証を行う。

4.5 Paging の書き換え

Xv6 では実メモリ (Physical memory) から仮想メモリ (Virtual memory) の変換を vm.c で行なっている。vm.c を CbC で書き換えていく。次の章で実際の書き換えについて説明する。

第5章　CbC インターフェース

Gears OS では Meta Code Gear で Context から値を取り出し、ノーマルレベルの Code Gear に値を渡す。しかし、Code Gaer がどの Data Gear の番号に対応するかを指定する必要があったり、 ノーマルレベルとメタレベルで見え方が異なる Data Gear を Meta Code Gear によって　調整する必要があったりと、 メタレベルからノーマルレベルの継続の記述が煩雑になるため、Interface 化をしている。Interface は Data Gear に対しての操作を行う Code Gear であり、実装は別で定義する。

Interface を使って記述することで Gears OS の機能を置き換えることできるようになる。

5.1　インターフェースの定義

インターフェースはある Data Gear の定義と、それに対する操作を行う Code Gear の集合を表現する Meta Data Gear である。Context では全ての Code Gaer と Data Gear の集合を表現していることに対し、インターフェースは一部の Code Gear と一部の Data Gear の集合を表現する。

インターフェースを記述することによってノーマルレベルとメタレベルの分離が可能となる。

Paging のインターフェース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: vm のインターフェースの定義 (vm.h)

```
1  typedef struct vm<Type,Impl> {
2      __code init_vmm(Impl* vm, __code next(...));
3      __code kpt_freerange(Impl* vm, uint low, uint hi, __code next(...));
4      __code kpt_alloc(Impl* vm ,__code next(...));
5      __code switchuvm(Impl* vm ,struct proc* p, __code next(...));
6      __code init_inituvm(Impl* vm, pde_t* pgdir, char* init, uint sz,
       __code next(...));
7      __code loaduvm(Impl* vm,pde_t* pgdir, char* addr, struct inode* ip,
       uint offset, uint sz,  __code next(...));
8      __code allocuvm(Impl* vm, pde_t* pgdir, uint oldsz, uint newsz,
       __code next(...));
9      __code clearpteu(Impl* vm, pde_t* pgdir, char* uva,  __code next(...)
       );
10     __code copyuvm(Impl* vm, pde_t* pgdir, uint sz, __code next(...));
11     __code uva2ka(Impl* vm, pde_t* pgdir, char* uva, __code next(...));
```

```
12      __code copyout(Impl* vm, pde_t* pgdir, uint va, void* pp, uint len,
        __code next(...));
13      __code paging_int(Impl* vm, uint phy_low, uint phy_hi, __code next
        (...));
14      __code void_ret(Impl* vm);
15      __code next(...);
16  } vm;
```

1 行目で実装名を定義している。typedef struct の直後に実装名 (vm) を書く。

Code Gear は__code CodeGearName () で記述する。第一引数である Impl* vm が Code Gear の型になる。

__code next(...) の引数 ... は複数の Input Data Gear を持つという意味である。後述する実装によって条件分岐によって複数の継続先が設定されることがある。それぞれの Code Gear の引数の 1 つに設定する。引数の最後に設定しているが遷移先で引数を受け取る順番が正しければよい。

Code Gaer は 2 行目から 15 行目のように "__code [Code Gear 名]([引数])"で定義する。この引数が input Data Gear になる。

インターフェースは Data Gear に対しての Code Gear とその Code Gear で扱われている Data Gear の集合を抽象化した Meta Data Gear で、vm.c に対応する実装は別で定義する。

5.2　インターフェースの実装の初期化

インターフェースの定義が終わったので次にインターフェースの使うための初期化についてソースコード 5.2 で示す。

ソースコード 5.2: vm インターフェースの初期化 (vm_impl.cbc)

```
1  #include "../../context.h"
2  #interface "vm.h"
3
4  vm* createvm_impl(struct Context* cbc_context) {
5      struct vm* vm  = new vm();
6      struct vm_impl* vm_impl = new vm_impl();
7      vm->vm = (union Data*)vm_impl;
8      vm_impl->vm_impl = NULL;
9      vm_impl->i  = 0;
10     vm_impl->pte = NULL;
11     vm_impl->sz  = 0;
12     vm_impl->loaduvm_ptesize_check = C_loaduvm_ptesize_checkvm_impl;
13     vm_impl->loaduvm_loop = C_loaduvm_loopvm_impl;
14     vm_impl->allocuvm_check_newsz = C_allocuvm_check_newszvm_impl;
15     vm_impl->allocuvm_loop = C_allocuvm_loopvm_impl;
16     vm_impl->copyuvm_check_null = C_copyuvm_check_nullvm_impl;
17     vm_impl->copyuvm_loop = C_copyuvm_loopvm_impl;
18     vm_impl->uva2ka_check_pe_types = C_uva2ka_check_pe_types;
```

```
19      vm_impl->paging_intvm_impl = C_paging_intvmvm_impl;
20      vm_impl->copyout_loopvm_impl = C_copyout_loopvm_impl;
21      vm_impl->switchuvm_check_pgdirvm_impl =
       C_switchuvm_check_pgdirvm_impl;
22      vm_impl->init_inituvm_check_sz = C_init_inituvm_check_sz;
23      vm->void_ret  = C_vm_void_ret;
24      vm->init_vmm = C_init_vmmvm_impl;
25      vm->kpt_freerange = C_kpt_freerangevm_impl;
26      vm->kpt_alloc = C_kpt_allocvm_impl;
27      vm->switchuvm = C_switchuvmvm_impl;
28      vm->init_inituvm = C_init_inituvmvm_impl;
29      vm->loaduvm = C_loaduvmvm_impl;
30      vm->allocuvm = C_allocuvmvm_impl;
31      vm->clearpteu = C_clearpteuvm_impl;
32      vm->copyuvm = C_copyuvmvm_impl;
33      vm->uva2ka = C_uva2kavm_impl;
34      vm->copyout = C_copyoutvm_impl;
35      vm->paging_int = C_paging_intvm_impl;
36      return vm;
37  }
```

2 行目のようにインターフェースのヘッダーファイルは #include ではなく #interface で呼び出す。

create_impl で、インターフェースを vm で定義する。5 行目で new vm() することでメモリ上にインターフェースの置き場所を確保。vm_impl も同じようにすることで実装の置き場所を確保し、7 行目でインターフェースと実装を紐付ける。

23 行目の vm->void_ret のようにそれぞれの Code Gear 名を enum の番号で代入していく。enum の番号を使う事で、ポインタによる誤ったアクセスを防ぐことができる。

5.3　インターフェース の実装

初期化したインターフェースの実装の例を説明する。ソースコード 5.2 の 24 行目 C_init_vmmvm_imp が 5.3 の 6 行目の init_vmmvm_impl に対応する。

ソースコード 5.3: vm インターフェースの使用 (vm_impl.cbc)

```
1   extern struct {
2       struct spinlock lock;
3       struct run *freelist;
4   } kpt_mem;
5
6   __code init_vmmvm_impl(struct vm_impl* vm,__code next(...)) {
7       initlock(&kpt_mem.lock, "vm");
8       kpt_mem.freelist = NULL;
9
10      goto next(...);
11  }
12
13  extern struct run {
```

```
14      struct run *next;
15  };
16
17  static void _kpt_free (char *v)
18  {
19      struct run *r;
20
21      r = (struct run*) v;
22      r->next = kpt_mem.freelist;
23      kpt_mem.freelist = r;
24  }
25
26  __code kpt_freerangevm_impl(struct vm_impl* vm, uint low, uint hi, __code
        next(...)) {
27
28     if (low < hi) {
29       _kpt_free((char*)low);
30       goto kpt_freerangevm_impl(vm, low + PT_SZ, hi, next(...));
31
32     }
33    goto next(...);
34  }
35
36  __code kpt_allocvm_impl(struct vm_impl* vm, __code next(...)) {
37    acquire(&kpt_mem.lock);
38
39    goto kpt_alloc_check_impl(vm_impl, next(...));
40  }
41
42  typedef struct proc proc;
43  __code switchuvmvm_impl(struct vm_impl* vm , struct proc* p, __code next
        (...)) { //:skip
44
45      goto switchuvm_check_pgdirvm_impl(...);
46  }
47
48  __code init_inituvmvm_impl(struct vm_impl* vm, pde_t* pgdir, char* init,
        uint sz, __code next(...)) {
49
50      Gearef(cbc_context, vm_impl)->pgdir = pgdir;
51      Gearef(cbc_context, vm_impl)->init = init;
52      Gearef(cbc_context, vm_impl)->sz = sz;
53      Gearef(cbc_context, vm_impl)->next = next;
54      goto init_inituvm_check_sz(vm, pgdir, init, sz, next(...));
55  }
56
57  __code loaduvmvm_impl(struct vm_impl* vm, pde_t* pgdir, char* addr,
        struct inode* ip, uint offset, uint sz,  __code next(...)) {
58      Gearef(cbc_context, vm_impl)->pgdir = pgdir;
59      Gearef(cbc_context, vm_impl)->addr = addr;
60      Gearef(cbc_context, vm_impl)->ip = ip;
61      Gearef(cbc_context, vm_impl)->offset = offset;
62      Gearef(cbc_context, vm_impl)->sz = sz;
```

```
63      Gearef(cbc_context, vm_impl)->next = next;
64
65      goto loaduvm_ptesize_checkvm_impl(vm, next(...));
66  }
67
68  __code allocuvmvm_impl(struct vm_impl* vm, pde_t* pgdir, uint oldsz, uint
        newsz, __code next(...)) {
69
70      goto allocuvm_check_newszvm_impl(vm, pgdir, oldsz, newsz, next(...));
71  }
72
73  __code clearpteuvm_impl(struct vm_impl* vm, pde_t* pgdir, char* uva,
        __code next(...)) {
74
75      goto clearpteu_check_ptevm_impl(vm, pgdir, uva, next(...));
76  }
77
78  __code copyuvmvm_impl(struct vm_impl* vm, pde_t* pgdir, uint sz, __code
        next(...)) {
79
80      goto copyuvm_check_nullvm_impl(vm, pgdir, sz, __code next(...));
81  }
82
83  __code uva2kavm_impl(struct vm_impl* vm, pde_t* pgdir, char* uva, __code
        next(...)) {
84
85      goto uva2ka_check_pe_types(vm, pgdir, uva, next(...));
86  }
87
88  __code copyoutvm_impl(struct vm_impl* vm, pde_t* pgdir, uint va, void* pp
        , uint len, __code next(...)) {
89
90      vm->buf = (char*) pp;
91
92      goto copyout_loopvm_impl(vm, pgdir, va, pp, len, va0, pa0, next(...))
        ;
93  }
94
95  __code paging_intvm_impl(struct vm_impl* vm, uint phy_low, uint phy_hi,
        __code next(...)) {
96
97      goto paging_intvmvm_impl(vm, phy_low, phy_hi, next(...));
98  }
99
100 __code vm_void_ret(struct vm_impl* vm) {
101     return;
102 }
```

5.4　インターフェース実装内の明示的な遷移の処理

CbC は状態遷移ベースで記述するため、for 文や if 文がある場合はさらに実装を分ける。

vm と同じように vm_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 (vm_impl_private.cbc)。

インターフェースの定義、実装、明示的な遷移の流れを図 5.1 に示す。

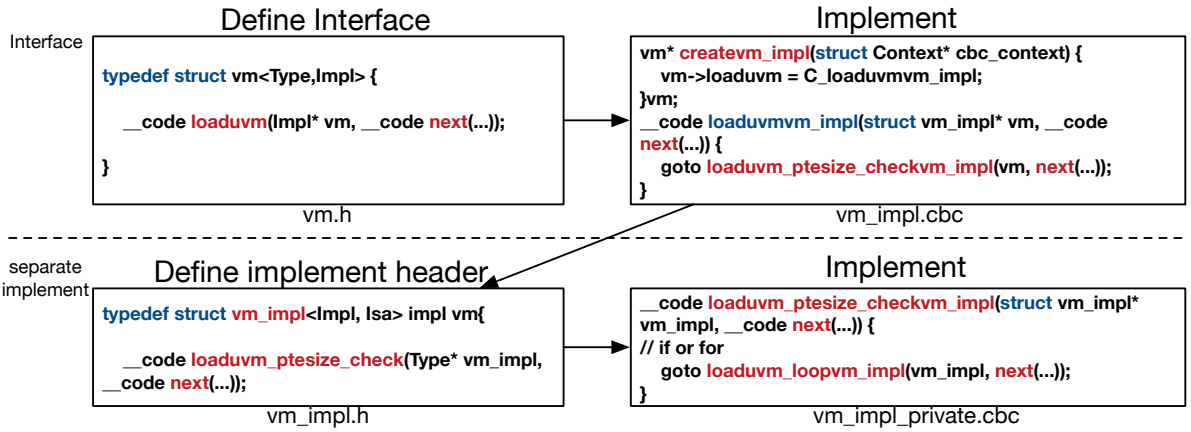

図 5.1: インターフェースの実装の流れ

インターフェースで定義した Code Gear 以外の Code Gaer も記述することができる。この Code Gear は基本的にインターフェースで指定された Code Gear 内からのみ継続されるため、Java の private メソッドのように扱われる。

インターフェースと同じようにヘッダーファイルをソースコード 5.4 で定義する。

ソースコード 5.4: vm private のヘッダーファイル

```
1  typedef struct vm_impl<Impl, Isa> impl vm{
2      union Data* vm_impl;
3      uint i;
4      pte_t* pte;
5      uint sz;
6      pde_t* pgdir;
7      char* addr;
8      struct inode* ip;
9      uint offset;
10     uint pa;
11     uint n;
12     uint oldsz;
13     uint newsz;
14     uint a;
15     int ret;
16     char* mem;
17     char* uva;
```

```
18      pde_t* d;
19      uint ap;
20      uint phy_low;
21      uint phy_hi;
22      uint va;
23      void* pp;
24      uint len;
25      char* buf;
26      char* pa0;
27      uint va0;
28      proc_struct* p;
29      char* init;
30
31      __code kpt_alloc_check_impl(Type* vm_impl, __code next(...));
32      __code loaduvm_ptesize_check(Type* vm_impl, __code next(int ret, ...)
        );
33      __code loaduvm_loop(Type* vm_impl, uint i, pte_t* pte, uint sz,
        __code next(int ret, ...));
34      __code allocuvm_check_newsz(Type* vm_impl, pde_t* pgdir, uint oldsz,
        uint newsz, __code next(...));
35      __code allocuvm_loop(Type* vm_impl, pde_t* pgdir, uint oldsz, uint
        newsz, uint a, __code next(...));
36      __code copyuvm_check_null(Type* vm_impl, pde_t* pgdir, uint sz,
        __code next(...));
37      __code copyuvm_loop(Type* vm_impl,pde_t* pgdir, uint sz, pde_t* d,
        pte_t* pte, uint pa, uint i, uint ap, char* mem, __code next(int ret,
        ...));
38      __code clearpteu_check_ptevm_impl(Type* vm_impl, pde_t* pgdir, char*
        uva,  __code next(...));
39      __code uva2ka_check_pe_types(Type* vm_impl, pde_t* pgdir, char* uva,
        __code next(...));
40      __code paging_intvm_impl(Type* vm_impl, uint phy_low, uint phy_hi,
        __code next(...));
41      __code copyout_loopvm_impl(Type* vm_impl, pde_t* pgdir, uint va, void
        * pp, uint len, __code next(...));
42      __code switchuvm_check_pgdirvm_impl(struct vm_impl* vm_impl, struct
        proc* p, __code next(...));
43      __code init_inituvm_check_sz(struct vm_impl* vm_impl, pde_t* pgdir,
        char* init, uint sz, __code next(...));
44      __code void_ret(Type* vm_impl);
45      __code next(...);
46  } vm_impl;
```

private での CbC の記述を vm.c と比べて説明する。全体の記述量が多いため、if 文と for 文のある loaduvm という関数で説明を行う。

ソースコード 5.5: vm.c の loaduvm

```
1  // Return the address of the PTE in page directory that corresponds to
2  // virtual address va.  If alloc!=0, create any required page table pages
       .
3  static pte_t* walkpgdir (pde_t *pgdir, const void *va, int alloc)
4  {
```

```
5       pde_t *pde;
6       pte_t *pgtab;
7
8       // pgdir points to the page directory, get the page direcotry entry (
        pde)
9       pde = &pgdir[PDE_IDX(va)];
10
11      if (*pde & PE_TYPES) {
12          pgtab = (pte_t*) p2v(PT_ADDR(*pde));
13
14      } else {
15          if (!alloc || (pgtab = (pte_t*) kpt_alloc()) == 0) {
16              return 0;
17          }
18
19          // Make sure all those PTE_P bits are zero.
20          memset(pgtab, 0, PT_SZ);
21
22          // The permissions here are overly generous, but they can
23          // be further restricted by the permissions in the page table
24          // entries, if necessary.
25          *pde = v2p(pgtab) | UPDE_TYPE;
26      }
27
28      return &pgtab[PTE_IDX(va)];
29  }
30
31  // Load a program segment into pgdir.  addr must be page-aligned
32  // and the pages from addr to addr+sz must already be mapped.
33  int loaduvm (pde_t *pgdir, char *addr, struct inode *ip, uint offset,
        uint sz)
34  {
35      uint i, pa, n;
36      pte_t *pte;
37
38      if ((uint) addr % PTE_SZ != 0) {
39          panic("loaduvm: addr must be page aligned");
40      }
41
42      for (i = 0; i < sz; i += PTE_SZ) {
43          if ((pte = walkpgdir(pgdir, addr + i, 0)) == 0) {
44              panic("loaduvm: address should exist");
45          }
46
47          pa = PTE_ADDR(*pte);
48
49          if (sz - i < PTE_SZ) {
50              n = sz - i;
51          } else {
52              n = PTE_SZ;
53          }
54
55          if (readi(ip, p2v(pa), offset + i, n) != n) {
```

```
56              return -1;
57          }
58      }
59
60      return 0;
61  }
```

vm_impl.cbc の Code Gear である loaduvmvm_impl から goto で loaduvm_ptesize_checkvm_impl に遷移する。vm.c での最初の if 文までの処理を 1 つの Code Gear として loaduvm_ptesize_checkvm_im に記述する。(3 行目 11 行目)

5.5　vm の Context

メモリ変換の処理で生成される Context の説明をする。code は全ての Code Gear を列挙した enum と関数ポインタの組みで表現される。(ソースコード 5.6 55 行目 68 行目) Code Gear の名前は enum で定義され、コンパイル後には整数で変換される。Code Gear に接続する際は enum で定義された番号を指定する。これによってメタ計算時に接続する Code Gear を切り替えることができる。

Data Gear のメモリ空間は事前に領域を確保した後、必要に応じて領域を割り当てることで実現する。実際に Allocation する際は ソースコード 5.6 の 9 行目で定義した heap を Data Gear のサイズ分増やすことで実現する。

ソースコード 5.6: 生成された Context

```
1  struct Context {
2      enum Code next;
3      struct Worker* worker;
4      struct TaskManager* taskManager;
5      int codeNum;
6      __code (**code) (struct Context*);
7      union Data **data;
8      void* heapStart;
9      void* heap;
10     long heapLimit;
11     int dataNum;
12
13     // task parameter
14     int idgCount; //number of waiting dataGear
15     int idg;
16     int maxIdg;
17     int odg;
18     int maxOdg;
19     int gpu; // GPU task
20     struct Context* task;
21     struct Element* taskList;
22 #ifdef USE_CUDAWorker
23     int num_exec;
24     CUmodule module;
```

```
25      CUfunction function;
26  #endif
27      /* multi dimension parameter */
28      int iterate;
29      struct Iterator* iterator;
30      enum Code before;
31  };
32
33  union Data {
34      ....
35      ///mnt/dalmore-home/one/src/cbcxv6/src/gearsTools/../interface/vm.h
36      struct vm {
37          union Data* vm;
38          uint low;
39          uint hi;
40          struct proc* p;
41          pde_t* pgdir;
42          char* init;
43          uint sz;
44          char* addr;
45          struct inode* ip;
46          uint offset;
47          uint oldsz;
48          uint newsz;
49          char* uva;
50          uint va;
51          void* pp;
52          uint len;
53          uint phy_low;
54          uint phy_hi;
55          enum Code init_vmm;
56          enum Code kpt_freerange;
57          enum Code kpt_alloc;
58          enum Code switchuvm;
59          enum Code init_inituvm;
60          enum Code loaduvm;
61          enum Code allocuvm;
62          enum Code clearpteu;
63          enum Code copyuvm;
64          enum Code uva2ka;
65          enum Code copyout;
66          enum Code paging_int;
67          enum Code void_ret;
68          enum Code next;
69      } vm;
70      ....
71  #ifndef CbC_XV6_CONTEXT
72      struct Context Context;
73  }; // union Data end       this is necessary for context generator
```

5.6　CbC のループ処理

CbC では goto での状態遷移によって実装するので loop は if 文を使って実装する。遷移図を 図 5.2 で示す。

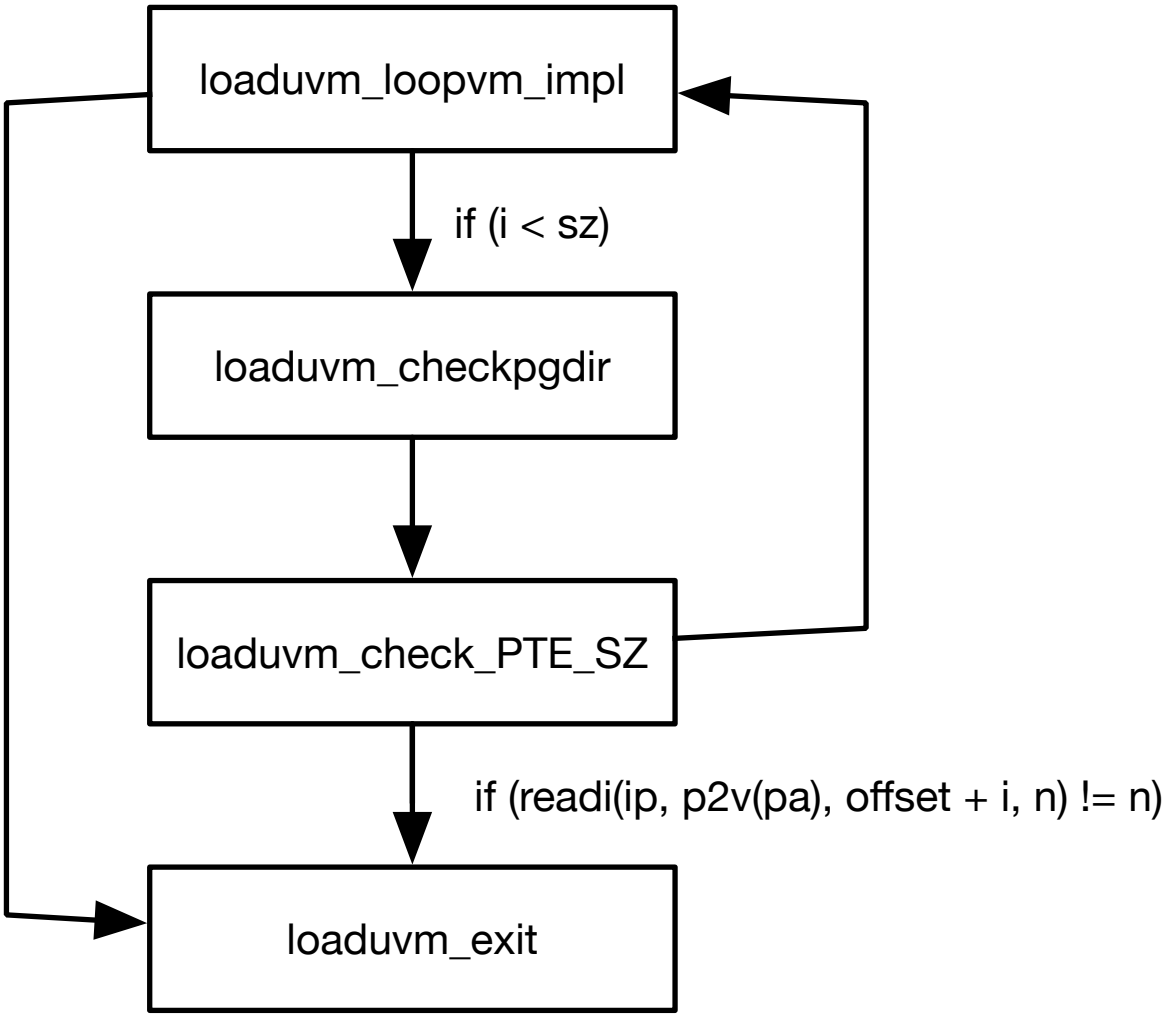

図 5.2: CbC によるループの実装

ソースコード 5.7: loaduvm の実装の記述

```
1 #interface "vm_impl.h"
2
3 __code loaduvm_ptesize_checkvm_impl(struct vm_impl* vm_impl,char* addr,
     __code next(int ret, ...)) {
4     if ((uint) addr %PTE_SZ != 0) {
```

```
5         char* msg = "addr % PTE_SZ != 0";
6         struct Err* err = createKernelError(&proc->cbc_context);
7         Gearef(cbc_context, Err)->msg = msg;
8         goto meta(cbc_context, err->panic);
9     }
10
11    goto loaduvm_loopvm_impl(vm_impl, next(ret, ...));
12 }
13
14 __code loaduvm_loopvm_impl(struct vm_impl* vm_impl, uint i, uint sz,
      __code next(int ret, ...)) {
15    if (i < sz) {
16        goto loaduvm_check_pgdir(vm_impl, next(ret, ...));
17    }
18
19    goto loaduvm_exit(vm_impl, next(ret, ...));
20 }
21
22
23 static pte_t* walkpgdir (pde_t *pgdir, const void *va, int alloc)
24 {
25    pde_t *pde;
26    pte_t *pgtab;
27
28    // pgdir points to the page directory, get the page direcotry entry (
      pde)
29    pde = &pgdir[PDE_IDX(va)];
30
31    if (*pde & PE_TYPES) {
32        pgtab = (pte_t*) p2v(PT_ADDR(*pde));
33
34    } else {
35        if (!alloc || (pgtab = (pte_t*) kpt_alloc()) == 0) {
36            return 0;
37        }
38
39        // Make sure all those PTE_P bits are zero.
40        memset(pgtab, 0, PT_SZ);
41
42        // The permissions here are overly generous, but they can
43        // be further restricted by the permissions in the page table
44        // entries, if necessary.
45        *pde = v2p(pgtab) | UPDE_TYPE;
46    }
47
48    return &pgtab[PTE_IDX(va)];
49 }
50
51
52 __code loaduvm_check_pgdir(struct vm_impl* vm_impl, pte_t* pte, pde_t*
      pgdir, uint i, char* addr, uint pa, __code next(int ret, ...)) {
53    if ((pte = walkpgdir(pgdir, addr + i, 0)) == 0) {
54        char* msg = "pte != walkpgdir...";
```

```
55          struct Err* err = createKernelError(&proc->cbc_context);
56          Gearef(cbc_context, Err)->msg = msg;
57          goto meta(cbc_context, err->panic);
58      }
59      pa = PTE_ADDR(*pte);
60
61      Gearef(cbc_context, vm_impl)->pte   = pte;
62      Gearef(cbc_context, vm_impl)->pgdir = pgdir;
63      Gearef(cbc_context, vm_impl)->addr  = addr;
64      Gearef(cbc_context, vm_impl)->pa    = pa;
65
66      goto loaduvm_check_PTE_SZ(vm_impl, next(ret, ...));
67  }
68
69  __code loaduvm_check_PTE_SZ(struct vm_impl* vm_impl, uint sz, uint i,
        uint n, struct inode* ip, uint pa, uint offset, __code next(int ret,
        ...)) {
70
71      if (sz - i < PTE_SZ) {
72          n = sz - i;
73      } else {
74          n = PTE_SZ;
75      }
76
77      if (readi(ip, p2v(pa), offset + i, n) != n) {
78          ret = -1;
79          goto next(ret, ...);
80      }
81
82      Gearef(cbc_context, vm_impl)->n = n;
83
84      goto loaduvm_loopvm_impl(vm_impl, next(ret, ...));
85  }
86
87  __code loaduvm_exit(struct vm_impl* vm_impl, __code next(int ret, ...)) {
88      ret = 0;
89      goto next(ret, ...);
90  }
```

for 文から末尾再起の変換について図 5.2 のように loaduvm_loopvm_impl の if 文でループの条件を書いて goto させ、loaduvm_check_PTE_SZ で満たしてなければ loaduvm_loopvm_impl に goto することでループを実装している。

static なものはまだ書き直していないが後々実装する

5.7 Meta Code Gear の記述

cmake によって生成されたメタ部分の記述について説明する。ファイルはビルドディレクトリ以下の /CMakeFiles/kernel.dir/c/ に生成される。

ノーマルレベルの Code Gear と ノーマルレベルの Code Gear 名 の後ろに _stub が付いた Meta Code Gear が対応する。例として loaduvm の生成された Code Gear と Meta Code Gear をソースコード 5.8 に示す。

ソースコード 5.8: loaduvm のメタ部分の記述

```
1  __code loaduvmvm_impl(struct Context *cbc_context,struct vm_impl* vm,
       pde_t* pgdir, char* addr, struct inode* ip, uint offset, uint sz,
       enum Code next) {
2      Gearef(cbc_context, vm_impl)->pgdir = pgdir;
3      Gearef(cbc_context, vm_impl)->addr = addr;
4      Gearef(cbc_context, vm_impl)->ip = ip;
5      Gearef(cbc_context, vm_impl)->offset = offset;
6      Gearef(cbc_context, vm_impl)->sz = sz;
7      Gearef(cbc_context, vm_impl)->next = next;
8
9      goto meta(cbc_context, C_loaduvm_ptesize_checkvm_impl);
10 }
11
12 __code loaduvmvm_impl_stub(struct Context* cbc_context) {
13         vm_impl* vm = (vm_impl*)GearImpl(cbc_context, vm, vm);
14         pde_t* pgdir = Gearef(cbc_context, vm)->pgdir;
15         char* addr = Gearef(cbc_context, vm)->addr;
16         inode* ip = Gearef(cbc_context, vm)->ip;
17         uint offset = Gearef(cbc_context, vm)->offset;
18         uint sz = Gearef(cbc_context, vm)->sz;
19         enum Code next = Gearef(cbc_context, vm)->next;
20         goto loaduvmvm_impl(cbc_context, vm, pgdir, addr, ip, offset, sz,
       next);
21 }
```

5.8　インターフェースの呼び出し

定義したインターフェースの呼び出し方について説明する。CbC の場合 goto による遷移を行うので、関数呼び出しのように goto 以降のコードを実行できない。

ソースコード 5.9: cbc インターフェースの goto

```
1  void userinit(void)
2  {
3      struct proc* p;
4      extern char _binary_initcode_start[], _binary_initcode_size[];
5
6      p = allocproc();
7      initContext(&p->cbc_context);
8
9      initproc = p;
10
11     if((p->pgdir = kpt_alloc()) == NULL) {
```

```
12          panic("userinit: out of memory?");
13      }
14
15      goto cbc_init_vmm_dummy(&p->cbc_context, p, p->pgdir,
        _binary_initcode_start, (int)_binary_initcode_size);
16      p->sz = PTE_SZ;
17
18      // craft the trapframe as if
19      memset(p->tf, 0, sizeof(*p->tf));
```

例として、ソースコード 5.9 の 15 行目のように goto によってインターフェースで定義した命令を行うと、戻ってこれないため 16 行目以降が実行されなくなる。

ソースコード 5.10: dummy を使った呼び出し

```
1  void cbc_init_vmm_dummy(struct Context* cbc_context, struct proc* p,
       pde_t* pgdir, char* init, uint sz)
2  {
3      // inituvm(p->pgdir, _binary_initcode_start, (int)
       _binary_initcode_size);
4
5      struct vm* vm = createvm_impl(cbc_context);
6      // goto vm->init_vmm(vm, pgdir, init, sz , vm->void_ret);
7          Gearef(cbc_context, vm)->vm = (union Data*) vm;
8          Gearef(cbc_context, vm)->pgdir = pgdir;
9          Gearef(cbc_context, vm)->init = init;
10         Gearef(cbc_context, vm)->sz = sz ;
11         Gearef(cbc_context, vm)->next = C_vm_void_ret ;
12     goto meta(cbc_context, vm->init_inituvm);
13 }
14
15 void userinit(void)
16 {
17     struct proc* p;
18     extern char _binary_initcode_start[], _binary_initcode_size[];
19
20     p = allocproc();
21     initContext(&p->cbc_context);
22
23     initproc = p;
24
25     if((p->pgdir = kpt_alloc()) == NULL) {
26         panic("userinit: out of memory?");
27     }
28
29     cbc_init_vmm_dummy(&p->cbc_context, p, p->pgdir,
       _binary_initcode_start, (int)_binary_initcode_size);
30
31     p->sz = PTE_SZ;
32 ....
33 }
```

7 行目から 11 行目の引数の設定に Gearef を使っているが、本来は CMake で生成しその

部分には何も書かない。11 行目の C_vm_void_ret は return するための enum コードであり、これを使って関数呼び出しのように振る舞う。

第6章　まとめ

OS 内部で CbC インターフェースを扱えるようになった。また、Linux 上で Arm のバイナリを吐けるように CMake を使った Cross Compile を行えるようになった。CbC の書き換えが完了すれば、継続の入力と出力を検査することで OS の信頼性を保証したり、インターフェースの実装の入れ替えが可能になる。

また、Context による複数環境の入れ替えや同時実行を可能にすることで CbCXv6 において コンテナと VM を実装ができると予想される。

謝辞

本研究と論文作成にあたり、ご多忙にも関わらず終始懇切なるご指導とご教授を賜わりました河野 真治准教授に心より感謝致します。また、先行研究として関わってくれた先輩方、共に研究を行い支えてくれた並列信頼研のメンバーに感謝いたします。最後に、理工学研究科情報工学専攻の学友、並びに家族に深く感謝いたします。

2020 年 3 月
桃原 優

参考文献

[1] 宮城光希. 継続を基本とした言語による os のモジュール化. Master's thesis, 琉球大学 大学院理工学研究科 情報工学専攻, 2019.

[2] 伊波立樹. Gears os の並列処理. Master's thesis, 琉球大学 大学院理工学研究科 情報工学専攻, 2018.

[3] Kaito TOKKMORI and Shinji KONO. Implementing continuation based language in llvm and clang. *LOLA 2015*, July 2015.

[4] Russ Cox, M Frans Kaashoek, and Robert Morris. Xv6, a simple unix-like teaching operating system, 2011.

[5] 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.

[6] 宮城光希, 河野真治. Code gear と data gear を持つ gears os の設計. 第 59 回プログラミング・シンポジウム, Jan 2018.

[7] 宮城光希, 河野真治. 継続を中心とした言語 gears os のデモンストレーション. 第 60 回プログラミング・シンポジウム, Jan 2019.

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

[9] 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.

[10] GNU Compiler Collection (GCC) Internals. *http://gcc.gnu.org/onlinedocs/gccint/*.

[11] Raspberry Pi — Teach, Learn, and Make with Raspberry Pi. https://www.raspberrypi.org.

[12] Zhiyi Wang. xv6-rpi. https://code.google.com/archive/p/xv6-rpi/, 2013.

[13] Hokama MASATAKA and Shinji KONO. 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, Jan 2019.

[14] Ulf Norell. Dependently typed programming in agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, pp. 1–2, New York, NY, USA, 2009. ACM.

[15] 青柳隆宏. はじめての OS コードリーディング –UNIX V6 で学ぶカーネルのしくみ. 2013.

[16] Herbert Bos Andrew S.Tanenbaum. Modern Operating Systems. 2015.

発表履歴

- 宮城 光希, 桃原 優, 河野真治. GearsOS のモジュール化と並列 API. 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, 2018
- 桃原 優, 東恩納琢偉, 河野真治. GearsOS の Paging と Segmentation・システムソフトウェアとオペレーティング・システム（OS）, May, 2019

]

付 録A　ソースコード一覧

本文中に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　インターフェースの 実装

Xv6 の元のコードである vm.c A.2 をインターフェースで定義した後に、if 文や for 文がある関数を実装側でさらに分けた vm_impl_private.cbc をソースコード A.1 に示す。

ソースコード A.1: vm の実装の private

```
1  #include "param.h"
2  #include "proc.h"
3  #include "mmu.h"
4  #include "defs.h"
5  #include "memlayout.h"
6  #interface "vm_impl.h"
7
8  /*
9  vm_impl* createvm_impl2(); //:skip
10 */
11
12 __code loaduvm_ptesize_checkvm_impl(struct vm_impl* vm_impl, __code next(
       int ret, ...)) {
13     char* addr = vm_impl->addr;
14
15     if ((uint) addr %PTE_SZ != 0) {
16        // goto panic
17     }
18
19     goto loaduvm_loopvm_impl(vm_impl, next(ret, ...));
20 }
21
22 __code loaduvm_loopvm_impl(struct vm_impl* vm_impl, __code next(int ret,
       ...)) {
23     uint i = vm_impl->i;
24     uint sz = vm_impl->sz;
25
26     if (i < sz) {
27         goto loaduvm_check_pgdir(vm_impl, next(ret, ...));
28     }
```

```
29 |
30 |     goto loaduvm_exit(vm_impl, next(ret, ...));
31 | }
32 |
33 |
34 | static pte_t* walkpgdir (pde_t *pgdir, const void *va, int alloc)
35 | {
36 |     pde_t *pde;
37 |     pte_t *pgtab;
38 |
39 |     // pgdir points to the page directory, get the page direcotry entry (
   |    pde)
40 |     pde = &pgdir[PDE_IDX(va)];
41 |
42 |     if (*pde & PE_TYPES) {
43 |         pgtab = (pte_t*) p2v(PT_ADDR(*pde));
44 |
45 |     } else {
46 |         if (!alloc || (pgtab = (pte_t*) kpt_alloc()) == 0) {
47 |             return 0;
48 |         }
49 |
50 |         // Make sure all those PTE_P bits are zero.
51 |         memset(pgtab, 0, PT_SZ);
52 |
53 |         // The permissions here are overly generous, but they can
54 |         // be further restricted by the permissions in the page table
55 |         // entries, if necessary.
56 |         *pde = v2p(pgtab) | UPDE_TYPE;
57 |     }
58 |
59 |     return &pgtab[PTE_IDX(va)];
60 | }
61 |
62 |
63 | __code loaduvm_check_pgdir(struct vm_impl* vm_impl, __code next(int ret,
   |     ...)) {
64 |     pte_t* pte = vm_impl->pte;
65 |     pde_t* pgdir = vm_impl->pgdir;
66 |     uint i = vm_impl->i;
67 |     char* addr = vm_impl->addr;
68 |     uint pa = vm_impl->pa;
69 |
70 |     if ((pte = walkpgdir(pgdir, addr + i, 0)) == 0) {
71 |         // goto panic
72 |     }
73 |     pa = PTE_ADDR(*pte);
74 |
75 |     vm_impl->pte = pte;
76 |     vm_impl->pgdir = pgdir;
77 |     vm_impl->addr = addr;
78 |     vm_impl->pa = pa;
79 |
80 |     goto loaduvm_check_PTE_SZ(vm_impl, next(ret, ...));
```

```
81 }
82
83 __code loaduvm_check_PTE_SZ(struct vm_impl* vm_impl, __code next(int ret,
       ...)) {
84     uint sz = vm_impl->sz;
85     uint i = vm_impl->i;
86     uint n = vm_impl->n;
87     struct inode* ip = vm_impl->ip;
88     uint pa = vm_impl->pa;
89     uint offset = vm_impl->offset;
90
91     if (sz - i < PTE_SZ) {
92         n = sz - i;
93     } else {
94         n = PTE_SZ;
95     }
96
97     if (readi(ip, p2v(pa), offset + i, n) != n) {
98         ret = -1;
99         goto next(ret, ...);
100    }
101
102    vm_impl->n = n;
103
104    goto loaduvm_loopvm_impl(vm_impl, next(ret, ...));
105 }
106
107 __code loaduvm_exit(struct vm_impl* vm_impl, __code next(int ret, ...)) {
108    ret = 0;
109    goto next(ret, ...);
110 }
111
112 struct run {
113    struct run *next;
114 };
115
116 struct {
117    struct spinlock lock;
118    struct run* freelist;
119 } kpt_mem;
120
121
122 static int mappages (pde_t *pgdir, void *va, uint size, uint pa, int ap)
123 {
124    char *a, *last;
125    pte_t *pte;
126
127    a = (char*) align_dn(va, PTE_SZ);
128    last = (char*) align_dn((uint)va + size - 1, PTE_SZ);
129
130    for (;;) {
131        if ((pte = walkpgdir(pgdir, a, 1)) == 0) {
132            return -1;
133        }
```

```
134 |
135 |         if (*pte & PE_TYPES) {
136 |             panic("remap");
137 |         }
138 |
139 |         *pte = pa | ((ap & 0x3) << 4) | PE_CACHE | PE_BUF | PTE_TYPE;
140 |
141 |         if (a == last) {
142 |             break;
143 |         }
144 |
145 |         a += PTE_SZ;
146 |         pa += PTE_SZ;
147 |     }
148 |
149 |     return 0;
150 | }
151 |
152 | __code kpt_alloc_check_impl(struct vm_impl* vm_impl, __code next(...)) {
153 |     struct run* r;
154 |     if ((r = kpt_mem.freelist) != NULL ) {
155 |         kpt_mem.freelist = r->next;
156 |     }
157 |     release(&kpt_mem.lock);
158 |
159 |     if ((r == NULL) && ((r = kmalloc (PT_ORDER)) == NULL)) {
160 |         // panic("oom: kpt_alloc");
161 |         // goto panic
162 |     }
163 |
164 |     memset(r, 0, PT_SZ);
165 |     goto next((char*)r);
166 | }
167 |
168 | __code allocuvm_check_newszvm_impl(struct vm_impl* vm_impl, pde_t* pgdir,
      |     uint oldsz, uint newsz, __code next(int ret, ...)){
169 |     if (newsz >= UADDR_SZ) {
170 |        goto next(0, ...);
171 |     }
172 |
173 |     if (newsz < oldsz) {
174 |        ret = newsz;
175 |        goto next(ret, ...);
176 |     }
177 |
178 |     char* mem;
179 |     uint a = align_up(oldsz, PTE_SZ);
180 |
181 |     goto allocuvm_loopvm_impl(vm_impl, pgdir, oldsz, newsz, mem, a, next(
      |    ret, ...));
182 | }
183 |
184 | __code allocuvm_loopvm_impl(struct vm_impl* vm_impl, pde_t* pgdir, uint
```

```
    oldsz, uint newsz, char* mem, uint a, __code next(int ret, ...)){
185
186     if (a < newsz) {
187         mem = alloc_page();
188
189         if (mem == 0) {
190             cprintf("allocuvm out of memory\n");
191             deallocuvm(pgdir, newsz, oldsz);
192             goto next(0, ...);
193         }
194
195         memset(mem, 0, PTE_SZ);
196         mappages(pgdir, (char*) a, PTE_SZ, v2p(mem), AP_KU);
197
198         goto allocuvm_loopvm_impl(vm_impl, pgdir, oldsz, newsz, a +
    PTE_SZ, next(ret, ...));
199     }
200     ret = newsz;
201     goto next(ret, ...);
202 }
203
204 __code clearpteu_check_ptevm_impl(struct vm_impl* vm_impl, pde_t* pgdir,
    char* uva, __code next(int ret, ...)) {
205     pte_t *pte;
206
207     pte = walkpgdir(pgdir, uva, 0);
208     if (pte == 0) {
209         // panic("clearpteu");
210         // goto panic;
211     }
212
213     // in ARM, we change the AP field (ap & 0x3) << 4)
214     *pte = (*pte & ~(0x03 << 4)) | AP_KO << 4;
215
216     goto next(ret, ...);
217 }
218
219 __code copyuvm_check_nullvm_impl(struct vm_impl* vm_impl, pde_t* pgdir,
    uint sz, __code next(int ret, ...)) {
220     pde_t *d;
221     pte_t *pte;
222     uint pa, i, ap;
223     char *mem;
224
225     // allocate a new first level page directory
226     d = kpt_alloc();
227     if (d == NULL ) {
228         ret = NULL;
229         goto next(ret, ...);
230     }
231     i = 0;
232
233     goto copyuvm_loopvm_impl(vm_impl, pgdir, sz, *d, *pte, pa, i, ap, *
```

```
    mem, next(ret, ...));
}

__code copyuvm_loopvm_impl(struct vm_impl* vm_impl, pde_t* pgdir, uint sz
    , pde_t* d, pte_t* pte, uint pa, uint i, uint ap, char* mem, __code
    next(int ret, ...)) {

     if (i < sz) {
         goto copyuvm_loop_check_walkpgdir(vm_impl, pgdir, sz, d, pte, pa,
     i, ap, mem, __code next(int ret, ...));

     }
     ret = d;
     goto next(ret, ...);
}

__code copyuvm_loop_check_walkpgdir(struct vm_impl* vm_impl, pde_t* pgdir
    , uint sz, pde_t* d, pte_t* pte, uint pa, uint i, uint ap, char* mem,
    __code next(int ret, ...)) {
         if ((pte = walkpgdir(pgdir, (void *) i, 0)) == 0) {
             // panic("copyuvm: pte should exist");
             // goto panic();
         }
     goto copyuvm_loop_check_pte(vm_impl, pgdir, sz, d, pte, pa, i, ap,
    mem, __code next(int ret, ...));
}

__code copyuvm_loop_check_pte(struct vm_impl* vm_impl, pde_t* pgdir, uint
     sz, pde_t* d, pte_t* pte, uint pa, uint i, uint ap, char* mem, __code
     next(int ret, ...)) {

         if (!(*pte & PE_TYPES)) {
             // panic("copyuvm: page not present");
             // goto panic();
         }

     goto copyuvm_loop_check_mem(vm_impl, pgdir, sz, d, pte, pa, i, ap,
    mem, __code next(int ret, ...));
}

__code copyuvm_loop_check_mem(struct vm_impl* vm_impl, pde_t* pgdir, uint
     sz, pde_t* d, pte_t* pte, uint pa, uint i, uint ap, char* mem, __code
     next(int ret, ...)) {
     pa = PTE_ADDR (*pte);
     ap = PTE_AP (*pte);

     if ((mem = alloc_page()) == 0) {
         goto copyuvm_loop_bad(vm_impl, d, next(...));
     }
     goto copyuvm_loop_check_mappages(vm_impl, pgdir, sz, d, pte, pa, i,
    ap, mem, __code next(int ret, ...));

}
```

```
274
275 __code copyuvm_loop_check_mappages(struct vm_impl* vm_impl, pde_t* pgdir,
        uint sz, pde_t* d, pte_t* pte, uint pa, uint i, uint ap, char* mem,
    __code next(int ret, ...)) {
276
277     memmove(mem, (char*) p2v(pa), PTE_SZ);
278
279     if (mappages(d, (void*) i, PTE_SZ, v2p(mem), ap) < 0) {
280         goto copyuvm_loop_bad(vm_impl, d, next(...));
281     }
282     goto copyuvm_loopvm_impl(vm_impl, pgdir, sz, d, pte, pa, i, ap, mem,
    __code next(int ret, ...));
283
284 }
285
286 __code copyuvm_loop_bad(struct vm_impl* vm_impl, pde_t* d, __code next(
    int ret, ...)) {
287     freevm(d);
288     ret = 0;
289     goto next(ret, ...);
290 }
291
292
293 __code uva2ka_check_pe_types(struct vm_impl* vm, pde_t* pgdir, char* uva,
        __code next(int ret, ...)) {
294     pte_t* pte;
295
296     pte = walkpgdir(pgdir, uva, 0);
297
298     // make sure it exists
299     if ((*pte & PE_TYPES) == 0) {
300         ret = 0;
301         goto next(ret, ...);
302     }
303     goto uva2ka_check_pte_ap(vm, pgdir, uva, pte, next(...));
304 }
305
306 __code uva2ka_check_pte_ap(struct vm_impl* vm, pde_t* pgdir, char* uva,
    pte_t* pte, __code next(int ret, ...)) {
307     // make sure it is a user page
308     if (PTE_AP(*pte) != AP_KU) {
309         ret = 0;
310         goto next(ret, ...);
311     }
312     ret = (char*) p2v(PTE_ADDR(*pte));
313     goto next(ret, ...);
314 }
315
316 // flush all TLB
317 static void flush_tlb (void)
318 {
319     uint val = 0;
320     asm("MCR p15, 0, %[r], c8, c7, 0" : :[r]"r" (val):);
```

```
321
322     // invalid entire data and instruction cache
323     asm ("MCR p15,0,%[r],c7,c10,0": :[r]"r" (val):);
324     asm ("MCR p15,0,%[r],c7,c11,0": :[r]"r" (val):);
325 }
326
327 __code paging_intvmvm_impl(struct vm_impl* vm_impl, uint phy_low, uint
        phy_hi, __code next(...)) {
328     mappages (P2V(&_kernel_pgtbl), P2V(phy_low), phy_hi - phy_low,
        phy_low, AP_KU);
329     flush_tlb ();
330
331     goto next(...);
332 }
333
334 __code copyout_loopvm_impl(struct vm_impl* vm_impl, pde_t* pgdir, uint va
        , void* pp, uint len, uint va0, char* pa0,  __code next(int ret, ...))
         {
335     if (len > 0) {
336         va0 = align_dn(va, PTE_SZ);
337         pa0 = uva2ka(pgdir, (char*) va0);
338         goto copyout_loop_check_pa0(vm_impl, pgdir, va, pp, len, va0, pa0
        , n, next(...));
339     }
340     ret = 0;
341     goto next(ret, ...);
342
343 }
344
345 __code copyout_loop_check_pa0(struct vm_impl* vm_impl, pde_t* pgdir, uint
         va, void* pp, uint len, uint va0, char* pa0, uint n, __code next(int
        ret, ...)) {
346     if (pa0 == 0) {
347         ret = -1;
348         goto next(ret, ...);
349     }
350     goto copyout_loop_check_n(vm_impl, pgdir, va, pp, len, va0, pa0, n,
        buf, next(...));
351 }
352 __code copyout_loop_check_n(struct vm_impl* vm_impl, pde_t* pgdir, uint
        va, void* pp, uint len, uint va0, char* pa0, uint n, char* buf, __code
         next(...)) {
353     n = PTE_SZ - (va - va0);
354
355     if (n > len) {
356         n = len;
357     }
358
359     len -= n;
360     buf += n;
361     va = va0 + PTE_SZ;
362     goto copyout_loopvm_impl(vm_impl, pgdir, va, pp, len, va0, pa0, next
        (...));
```

```
363 }
364
365 typedef struct proc proc_struct;
366 __code switchuvm_check_pgdirvm_impl(struct vm_impl* vm_impl, proc_struct*
        p, __code next(...)) { //:skip
367     uint val;
368
369     pushcli();
370
371     if (p->pgdir == 0) {
372         panic("switchuvm: no pgdir");
373     }
374
375     val = (uint) V2P(p->pgdir) | 0x00;
376
377     asm("MCR p15, 0, %[v], c2, c0, 0": :[v]"r" (val):);
378     flush_tlb();
379
380     popcli();
381
382     goto next(...);
383 }
384
385 __code init_inituvm_check_sz(struct vm_impl* vm_impl, pde_t* pgdir, char*
        init, uint sz, __code next(...)) {
386     char* mem;
387
388     if (sz >= PTE_SZ) {
389         // goto panic;
390         // panic("inituvm: more than a page");
391     }
392
393     mem = alloc_page();
394     memset(mem, 0, PTE_SZ);
395     mappages(pgdir, 0, PTE_SZ, v2p(mem), AP_KU);
396     memmove(mem, init, sz);
397
398     goto next(...);
399 }
```

A-2 Xv6 の Paging

Paging の記述である Xv6 の vm.c のコードを A.2 に示す。

ソースコード A.2: Xv6 の vm.c

```
1 #include "param.h"
2 #include "types.h"
3 #include "defs.h"
4 #include "arm.h"
5 #include "memlayout.h"
```

```
6  #include "mmu.h"
7  #include "proc.h"
8  #include "spinlock.h"
9  #include "elf.h"
10
11 extern char data[];  // defined by kernel.ld
12 pde_t *kpgdir;  // for use in scheduler()
13
14 // Xv6 can only allocate memory in 4KB blocks. This is fine
15 // for x86. ARM's page table and page directory (for 28-bit
16 // user address) have a size of 1KB. kpt_alloc/free is used
17 // as a wrapper to support allocating page tables during boot
18 // (use the initial kernel map, and during runtime, use buddy
19 // memory allocator.
20 struct run {
21     struct run *next;
22 };
23
24 struct {
25     struct spinlock lock;
26     struct run *freelist;
27 } kpt_mem;
28
29 void init_vmm (void)
30 {
31     initlock(&kpt_mem.lock, "vm");
32     kpt_mem.freelist = NULL;
33 }
34
35 static void _kpt_free (char *v)
36 {
37     struct run *r;
38
39     r = (struct run*) v;
40     r->next = kpt_mem.freelist;
41     kpt_mem.freelist = r;
42 }
43
44
45 static void kpt_free (char *v)
46 {
47     if (v >= (char*)P2V(INIT_KERNMAP)) {
48         kfree(v, PT_ORDER);
49         return;
50     }
51
52     acquire(&kpt_mem.lock);
53     _kpt_free (v);
54     release(&kpt_mem.lock);
55 }
56
57 // add some memory used for page tables (initialization code)
58 void kpt_freerange (uint32 low, uint32 hi)
```

```
59 {
60     while (low < hi) {
61         _kpt_free ((char*)low);
62         low += PT_SZ;
63     }
64 }
65
66 void* kpt_alloc (void)
67 {
68     struct run *r;
69
70     acquire(&kpt_mem.lock);
71
72     if ((r = kpt_mem.freelist) != NULL ) {
73         kpt_mem.freelist = r->next;
74     }
75
76     release(&kpt_mem.lock);
77
78     // Allocate a PT page if no inital pages is available
79     if ((r == NULL) && ((r = kmalloc (PT_ORDER)) == NULL)) {
80         panic("oom: kpt_alloc");
81     }
82
83     memset(r, 0, PT_SZ);
84     return (char*) r;
85 }
86
87 // Return the address of the PTE in page directory that corresponds to
88 // virtual address va.  If alloc!=0, create any required page table pages
       .
89 static pte_t* walkpgdir (pde_t *pgdir, const void *va, int alloc)
90 {
91     pde_t *pde;
92     pte_t *pgtab;
93
94     // pgdir points to the page directory, get the page direcotry entry (
       pde)
95     pde = &pgdir[PDE_IDX(va)];
96
97     if (*pde & PE_TYPES) {
98         pgtab = (pte_t*) p2v(PT_ADDR(*pde));
99
100    } else {
101        if (!alloc || (pgtab = (pte_t*) kpt_alloc()) == 0) {
102            return 0;
103        }
104
105        // Make sure all those PTE_P bits are zero.
106        memset(pgtab, 0, PT_SZ);
107
108        // The permissions here are overly generous, but they can
109        // be further restricted by the permissions in the page table
```

```
110         // entries, if necessary.
111         *pde = v2p(pgtab) | UPDE_TYPE;
112     }
113
114     return &pgtab[PTE_IDX(va)];
115 }
116
117 // Create PTEs for virtual addresses starting at va that refer to
118 // physical addresses starting at pa. va and size might not
119 // be page-aligned.
120 static int mappages (pde_t *pgdir, void *va, uint size, uint pa, int ap)
121 {
122     char *a, *last;
123     pte_t *pte;
124
125     a = (char*) align_dn(va, PTE_SZ);
126     last = (char*) align_dn((uint)va + size - 1, PTE_SZ);
127
128     for (;;) {
129         if ((pte = walkpgdir(pgdir, a, 1)) == 0) {
130             return -1;
131         }
132
133         if (*pte & PE_TYPES) {
134             panic("remap");
135         }
136
137         *pte = pa | ((ap & 0x3) << 4) | PE_CACHE | PE_BUF | PTE_TYPE;
138
139         if (a == last) {
140             break;
141         }
142
143         a += PTE_SZ;
144         pa += PTE_SZ;
145     }
146
147     return 0;
148 }
149
150 // flush all TLB
151 static void flush_tlb (void)
152 {
153     uint val = 0;
154     asm("MCR p15, 0, %[r], c8, c7, 0" : :[r]"r" (val):);
155
156     // invalid entire data and instruction cache
157     asm ("MCR p15,0,%[r],c7,c10,0": :[r]"r" (val):);
158     asm ("MCR p15,0,%[r],c7,c11,0": :[r]"r" (val):);
159 }
160
161 // Switch to the user page table (TTBR0)
162 void switchuvm (struct proc *p)
```

```
163 {
164     uint val;
165 
166     pushcli();
167 
168     if (p->pgdir == 0) {
169         panic("switchuvm: no pgdir");
170     }
171 
172     val = (uint) V2P(p->pgdir) | 0x00;
173 
174     asm("MCR p15, 0, %[v], c2, c0, 0": :[v]"r" (val):);
175     flush_tlb();
176 
177     popcli();
178 }
179 
180 // Load the initcode into address 0 of pgdir. sz must be less than a page
    .
181 void inituvm (pde_t *pgdir, char *init, uint sz)
182 {
183     char *mem;
184 
185     if (sz >= PTE_SZ) {
186         panic("inituvm: more than a page");
187     }
188 
189     mem = alloc_page();
190     memset(mem, 0, PTE_SZ);
191     mappages(pgdir, 0, PTE_SZ, v2p(mem), AP_KU);
192     memmove(mem, init, sz);
193 }
194 
195 // Load a program segment into pgdir.  addr must be page-aligned
196 // and the pages from addr to addr+sz must already be mapped.
197 int loaduvm (pde_t *pgdir, char *addr, struct inode *ip, uint offset,
    uint sz)
198 {
199     uint i, pa, n;
200     pte_t *pte;
201 
202     if ((uint) addr % PTE_SZ != 0) {
203         panic("loaduvm: addr must be page aligned");
204     }
205 
206     for (i = 0; i < sz; i += PTE_SZ) {
207         if ((pte = walkpgdir(pgdir, addr + i, 0)) == 0) {
208             panic("loaduvm: address should exist");
209         }
210 
211         pa = PTE_ADDR(*pte);
212 
213         if (sz - i < PTE_SZ) {
```

```
214             n = sz - i;
215         } else {
216             n = PTE_SZ;
217         }
218 
219         if (readi(ip, p2v(pa), offset + i, n) != n) {
220             return -1;
221         }
222     }
223 
224     return 0;
225 }
226 
227 // Allocate page tables and physical memory to grow process from oldsz to
228 // newsz, which need not be page aligned.  Returns new size or 0 on error
    .
229 int allocuvm (pde_t *pgdir, uint oldsz, uint newsz)
230 {
231     char *mem;
232     uint a;
233 
234     if (newsz >= UADDR_SZ) {
235         return 0;
236     }
237 
238     if (newsz < oldsz) {
239         return oldsz;
240     }
241 
242     a = align_up(oldsz, PTE_SZ);
243 
244     for (; a < newsz; a += PTE_SZ) {
245         mem = alloc_page();
246 
247         if (mem == 0) {
248             cprintf("allocuvm out of memory\n");
249             deallocuvm(pgdir, newsz, oldsz);
250             return 0;
251         }
252 
253         memset(mem, 0, PTE_SZ);
254         mappages(pgdir, (char*) a, PTE_SZ, v2p(mem), AP_KU);
255     }
256 
257     return newsz;
258 }
259 
260 // Deallocate user pages to bring the process size from oldsz to
261 // newsz.  oldsz and newsz need not be page-aligned, nor does newsz
262 // need to be less than oldsz.  oldsz can be larger than the actual
263 // process size.  Returns the new process size.
264 int deallocuvm (pde_t *pgdir, uint oldsz, uint newsz)
265 {
266     pte_t *pte;
```

```
267     uint a;
268     uint pa;
269
270     if (newsz >= oldsz) {
271         return oldsz;
272     }
273
274     for (a = align_up(newsz, PTE_SZ); a < oldsz; a += PTE_SZ) {
275         pte = walkpgdir(pgdir, (char*) a, 0);
276
277         if (!pte) {
278             // pte == 0 --> no page table for this entry
279             // round it up to the next page directory
280             a = align_up (a, PDE_SZ);
281
282         } else if ((*pte & PE_TYPES) != 0) {
283             pa = PTE_ADDR(*pte);
284
285             if (pa == 0) {
286                 panic("deallocuvm");
287             }
288
289             free_page(p2v(pa));
290             *pte = 0;
291         }
292     }
293
294     return newsz;
295 }
296
297 // Free a page table and all the physical memory pages
298 // in the user part.
299 void freevm (pde_t *pgdir)
300 {
301     uint i;
302     char *v;
303
304     if (pgdir == 0) {
305         panic("freevm: no pgdir");
306     }
307
308     // release the user space memroy, but not page tables
309     deallocuvm(pgdir, UADDR_SZ, 0);
310
311     // release the page tables
312     for (i = 0; i < NUM_UPDE; i++) {
313         if (pgdir[i] & PE_TYPES) {
314             v = p2v(PT_ADDR(pgdir[i]));
315             kpt_free(v);
316         }
317     }
318
319     kpt_free((char*) pgdir);
```

```
320 }
321
322 // Clear PTE_U on a page. Used to create an inaccessible page beneath
323 // the user stack (to trap stack underflow).
324 void clearpteu (pde_t *pgdir, char *uva)
325 {
326     pte_t *pte;
327
328     pte = walkpgdir(pgdir, uva, 0);
329     if (pte == 0) {
330         panic("clearpteu");
331     }
332
333     // in ARM, we change the AP field (ap & 0x3) << 4)
334     *pte = (*pte & ~(0x03 << 4)) | AP_KO << 4;
335 }
336
337 // Given a parent process's page table, create a copy
338 // of it for a child.
339 pde_t* copyuvm (pde_t *pgdir, uint sz)
340 {
341     pde_t *d;
342     pte_t *pte;
343     uint pa, i, ap;
344     char *mem;
345
346     // allocate a new first level page directory
347     d = kpt_alloc();
348     if (d == NULL ) {
349         return NULL ;
350     }
351
352     // copy the whole address space over (no COW)
353     for (i = 0; i < sz; i += PTE_SZ) {
354         if ((pte = walkpgdir(pgdir, (void *) i, 0)) == 0) {
355             panic("copyuvm: pte should exist");
356         }
357
358         if (!(*pte & PE_TYPES)) {
359             panic("copyuvm: page not present");
360         }
361
362         pa = PTE_ADDR (*pte);
363         ap = PTE_AP (*pte);
364
365         if ((mem = alloc_page()) == 0) {
366             goto bad;
367         }
368
369         memmove(mem, (char*) p2v(pa), PTE_SZ);
370
371         if (mappages(d, (void*) i, PTE_SZ, v2p(mem), ap) < 0) {
372             goto bad;
```

```
373         }
374     }
375     return d;
376
377 bad: freevm(d);
378     return 0;
379 }
380
381 //PAGEBREAK!
382 // Map user virtual address to kernel address.
383 char* uva2ka (pde_t *pgdir, char *uva)
384 {
385     pte_t *pte;
386
387     pte = walkpgdir(pgdir, uva, 0);
388
389     // make sure it exists
390     if ((*pte & PE_TYPES) == 0) {
391         return 0;
392     }
393
394     // make sure it is a user page
395     if (PTE_AP(*pte) != AP_KU) {
396         return 0;
397     }
398
399     return (char*) p2v(PTE_ADDR(*pte));
400 }
401
402 // Copy len bytes from p to user address va in page table pgdir.
403 // Most useful when pgdir is not the current page table.
404 // uva2ka ensures this only works for user pages.
405 int copyout (pde_t *pgdir, uint va, void *p, uint len)
406 {
407     char *buf, *pa0;
408     uint n, va0;
409
410     buf = (char*) p;
411
412     while (len > 0) {
413         va0 = align_dn(va, PTE_SZ);
414         pa0 = uva2ka(pgdir, (char*) va0);
415
416         if (pa0 == 0) {
417             return -1;
418         }
419
420         n = PTE_SZ - (va - va0);
421
422         if (n > len) {
423             n = len;
424         }
425
426         memmove(pa0 + (va - va0), buf, n);
```

```
427
428         len -= n;
429         buf += n;
430         va = va0 + PTE_SZ;
431     }
432
433     return 0;
434 }
435
436
437 // 1:1 map the memory [phy_low, phy_hi] in kernel. We need to
438 // use 2-level mapping for this block of memory. The rumor has
439 // it that ARMv6's small brain cannot handle the case that memory
440 // be mapped in both 1-level page table and 2-level page. For
441 // initial kernel, we use 1MB mapping, other memory needs to be
442 // mapped as 4KB pages
443 void paging_init (uint phy_low, uint phy_hi)
444 {
445     mappages (P2V(&_kernel_pgtbl), P2V(phy_low), phy_hi - phy_low,
    phy_low, AP_KU);
446     flush_tlb ();
447 }
```