

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 を要する機器に依存している現代では、OS のバグは日常生活に支障を来たすことに繋がる。OS 自体に信頼性が求められるが、機能が多く短期間のアップデートが必要な OS では、全てのコードに対して何度も検証を行うのは困難である。

本研究室で開発したメタレベルの記述ができる CbC という言語を用いて OS の信頼性を保証するために Gears OS を開発中である。本研究での Gears OS の実装は、Xv6 という OS を参考に行っている。CbC は継続を中心とした言語であるため、状態遷移モデルに落とし込むことができる。状態遷移の 1 つ 1 つの関数に対して検証を行うことで OS の機能の信頼性を保証することができる。

しかし、ノーマルレベルとメタレベルで見え方が異なるデータに対して継続の記述が煩雑になる。そこで本研究では、インターフェースを用いたモジュール化を行う。インターフェースによって形式化され柔軟に再利用することもできる。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と考察を行う。

# Abstract

OS Bugs lead to hinder daily life, In the modern times of relying on devices that require an OS. OS requires reliability. But It is defficult to verify all the code many times on an OS that has meny functions and requires short-term updates.

Our laboratory developing an OS called Gears OS to guarantee reliability. Gears OS uses a programming language called CbC that can write meta-level. GearsOS implementation is based on Xv6. CbC is a language with a focus on continuation, it can be dropped into a state transition model. The reliability of OS functions can be guaranteed by verifying each function of the state transition. But, The description of continuation becomes complicated for data that looks defferent between the normal-level and the meta-level. In this research, mojularization using interfaces. Interface formalized and can be flexibly reused. This paper describes and discusses the interface and implementation of the memory management part, which is the basis of OS reliability.

# 目次

第1章 OS の信頼性の保障	2
第2章 CbC による Geas OS の開発	4
2.1 Code Gear と Data Gear	4
2.2 Meta Code Gear と Meta Data Gear	5
2.3 Context	5
第3章 Xv6	8
3.1 Kernel Space と User Space	8
3.2 system call	8
3.3 Xv6-rpi	9
第4章 CbCXv6 での Paging	10
4.1 Xv6 を元にした Gears OS の実装	10
4.2 Paging	10
4.3 CbCXv6 での Paging	10
4.4 Paging の書き換え	11
第5章 CbC インターフェース	13
5.1 インターフェースの定義	13
5.2 インターフェースの実装	14
5.3 インターフェース内の private メソッド	18
5.4 インターフェースの呼び出し	23
第6章 評価	26
第7章 まとめ	27
7.1 今後の書き換え方針	27
謝辞	27

参考文献	29
発表履歴	30
付録	31
付録 A ソースコード一覧	32
A-1 インターフェース内の private メソッドの実装 . . . . .	32

# 目 次

2.1	Code Gear 間の継続 . . . . .	4
2.2	ノーマルレベルとメタレベルの継続の見え方 . . . . .	5
4.1	On the left, xv6's kernel address space. RWX refer to PTE read, write, and execute permissions. On the right, the RISC-V physical address space that xv6 expects to see. Russ Cox(2014) xv6 a simple, Unix-like teaching operating system (Frans Kaashoek, Robert Morris) . . . . .	12
5.1	CbC によるループの実装 . . . . .	25

# 表 目 次



# ソースコード目次

2.1	生成された Context . . . . .	6
3.1	xv6 のシステムコールのリスト . . . . .	8
5.1	vm のインターフェースの定義 (vm.h) . . . . .	13
5.2	vm インターフェースの実装 . . . . .	15
5.3	vm private のヘッダーファイル . . . . .	18
5.4	vm.c の loadvm . . . . .	19
5.5	private での loadvm の実装 . . . . .	20
5.6	cbc インターフェースの goto . . . . .	23
5.7	dummy を使った呼び出し . . . . .	23
A.1	Xv6 の vm.c . . . . .	32
A.2	vm の実装の private . . . . .	40

# 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 にアクセスできるバグや、コンピュータの日付の設定を変更するとコンピュータ自体が壊れるバグが発生した。

OS 自体に信頼性が求められるが、複雑な機能が多く、短期間のアップデートが必要な OS では、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 が行なっているため、ユーザーが信頼性を保証することもできない。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。OS による資源管理によってユーザーは資源を気にすることなくコンピュータを扱うことができる。

このように OS には資源管理やシステムコールされた後の処理などユーザーが記述するプログラム以外の部分が存在する。その処理をメタレベルの計算、ユーザーが記述する部分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 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、状態遷移モデルに落とし込むことができる。Code Gear に対して入力 of Data Gear と出力 of Data Gear が存在し、入力に対して期待される出力がされてるか検査することで信頼性を保証する。

モデル検査には定理証明支援系である Agda を用いる。Agda は Haure Logic という検証手法を扱うことができる。Haure Logic は事前条件を使ってある関数を実行して事後条件を満たすことを確認する検証手法であり、継続に事前条件と事後条件を持たせることのできる CbC と相性がいい。

CbC を使って 信頼性の保証と拡張性を持たせる Gears OS の開発を行なっている。本論文では、Xv6 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 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が可能となる。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の違いが生じ、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した。インターフェースを使うことで機能の入れ替えによる拡張性や Agda による証明が可能となることを目的する。

## 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。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。

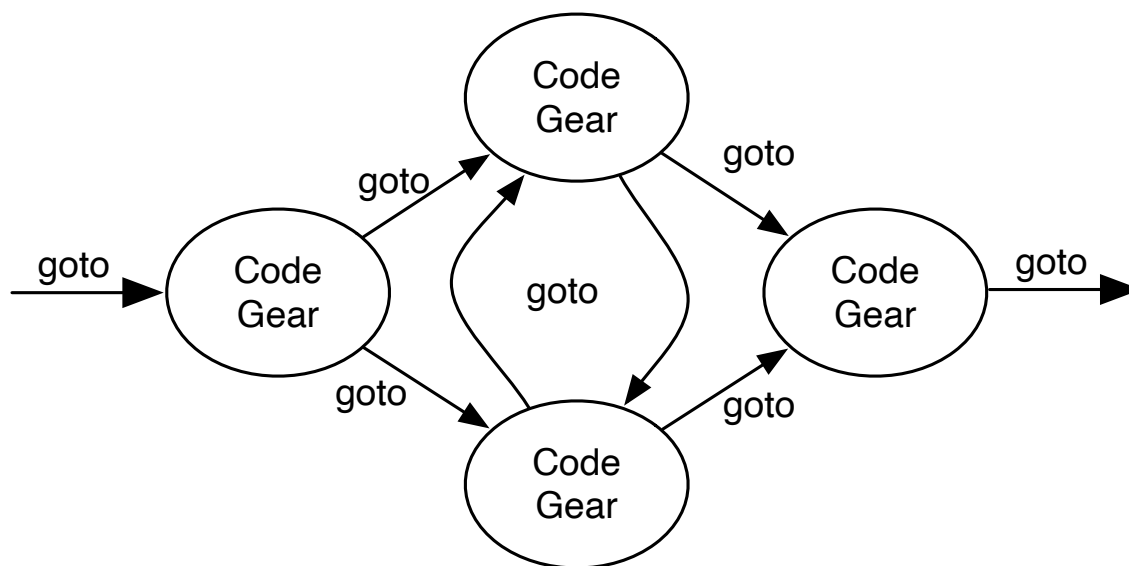


図 2.1: Code Gear 間の継続

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。メタレベルの記述によって User Space 側からメモリ管理を行えるようになる。

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。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 5.1 の下のように継続を行っている。

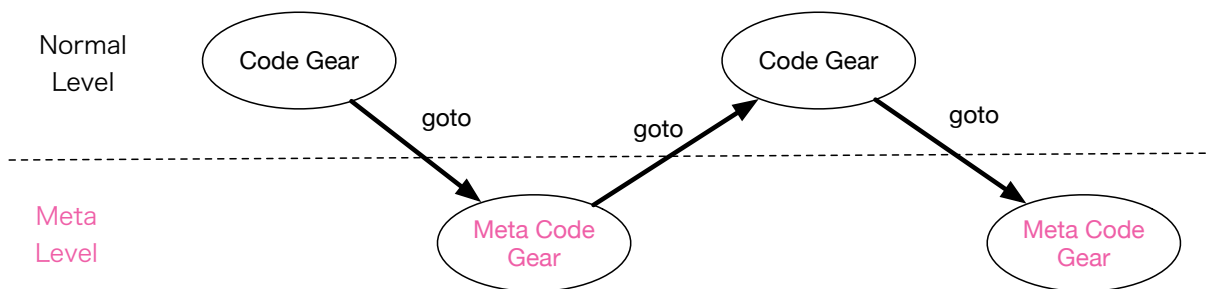


図 2.2: ノーマルレベルとメタレベルの継続の見え方

## 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。従来のスレッドやプロセスに対応する。Gears OS では Code Gear と Data Gear への接続を Context を通して行う。Context が持つ Data Gear のメモリ空間は事前に確保され、Data Gear のメモリ確保の際に heap の値をずらしてメモリを割り当てる。

ノーマルレベルの Code Gear から Meta Data Gear である Context を直接参照してしまうと、ユーザーがメタ計算をノーマルレベルで自由に記述できてしまい、メタ計算を分

離した意味がなくなってしまう。この問題を防ぐため、Context から必要な Data Gear のみをノーマルレベルの Code Gear に渡す処理を行なっている。

Meta Code Gear は使用される全ての Code Gear ごとに記述する必要がある。しかし、全ての Code Gear に対して記述すると膨大な記述量になる。そのため、Interface を実装した code Gear の Meta Code Gear は Perl スクリプトで自動生成する。

Meta Code Gear はユーザーが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。

第 5 章で扱うメモリ管理部分である vm の Context を ソースコード 2.1 に示す。

ソースコード 2.1: 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 switchvm;
59     enum Code init_initvm;
60     enum Code loadvm;
61     enum Code allocvm;
62     enum Code clearpteu;
63     enum Code copyvm;
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
```

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 2.19 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。

## 第3章 Xv6

Xv6 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 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 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 という OS を用意して動作しているか検証中である。xv6-rpi は CbCXv6 と別で用意している。

## 第4章 CbCXv6 での Paging

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。メモリ管理の手法に、Paging がある。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。

図 4.1 で Xv6 の仮想メモリと実メモリについて説明する。図の RWX は読み込み、書き込み、実行の権限を表している。

### 4.3 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。

## 4.4 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。次のチャプターで実際の書き換えについて説明する。

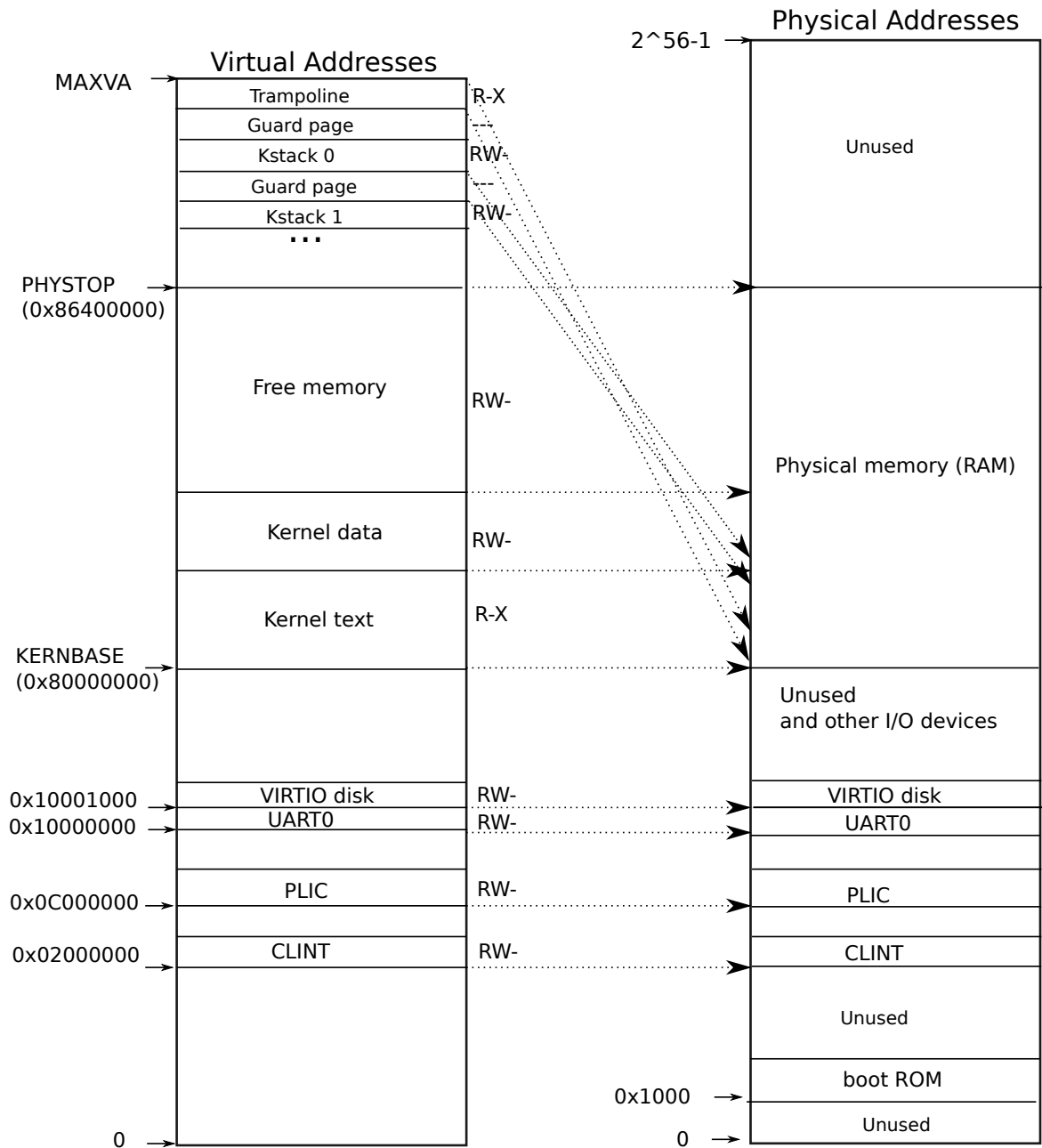


図 4.1: On the left, xv6's kernel address space. RWX refer to PTE read, write, and execute permissions. On the right, the RISC-V physical address space that xv6 expects to see. Russ Cox(2014) xv6 a simple, Unix-like teaching operating system (Frans Kaashoek, Robert Morris)

## 第5章 CbC インターフェース

構造図書く (今の cbcxv6 と同じか確認してから)

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。

そうすることで Gears OS の機能を置き換えることができるようになる。

### 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 switchvm(Impl* vm, struct proc* p, __code next(...));
6     __code init_initvm(Impl* vm, pde_t* pgdir, char* init, uint sz,
7         __code next(...));
8     __code loadvm(Impl* vm, pde_t* pgdir, char* addr, struct inode* ip,
9         uint offset, uint sz, __code next(...));
10    __code allocvm(Impl* vm, pde_t* pgdir, uint oldsz, uint newsz,
11        __code next(...));
12    __code clearpteu(Impl* vm, pde_t* pgdir, char* uva, __code next(...));
13    };
```

```

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,
13 |     __code next(...));
14 |     __code paging_int(Impl* vm, uint phy_low, uint phy_hi, __code next
15 |     (...));
16 |     __code void_ret(Impl* vm);
17 |     __code next(...);
18 | } 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。

2 行目から 19 行目で引数の Data Gear 郡を定義している。初期化された Data Gear がそれぞれの Code Gear の引数として扱われる。例として、2 行目で定義された vm が 21 行目から 32 行目までの引数と対応している。

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 20 行目から 33 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。

## 5.2 インターフェースの実装

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。

インターフェースの実装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実装

```

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 | }
38 | extern struct {
39 |     struct spinlock lock;
40 |     struct run *freelist;
41 | } kpt_mem;
42 |
43 | __code init_vmmvm_impl(struct vm_impl* vm, __code next(...)) {
44 |     initlock(&kpt_mem.lock, "vm");
45 |     kpt_mem.freelist = NULL;
46 |
47 |     goto next(...);
48 | }
49 |
50 | extern struct run {
51 |     struct run *next;
52 | };
53 |
54 | static void _kpt_free (char *v)
55 | {
56 |     struct run *r;
57 |
58 |     r = (struct run*) v;
59 |     r->next = kpt_mem.freelist;
60 |     kpt_mem.freelist = r;
61 | }
62 |
63 | __code kpt_freerangevm_impl(struct vm_impl* vm, uint low, uint hi, __code
    | next(...)) {
64 |
65 |     if (low < hi) {
66 |         _kpt_free((char*)low);

```

```

67     goto kpt_freerangevm_impl(vm, low + PT_SZ, hi, next(...));
68 }
69 }
70 goto next(...);
71 }
72
73 __code kpt_allocvm_impl(struct vm_impl* vm, __code next(...)) {
74     acquire(&kpt_mem.lock);
75
76     goto kpt_alloc_check_impl(vm_impl, next(...));
77 }
78
79 typedef struct proc proc;
80 __code switchvmvm_impl(struct vm_impl* vm , struct proc* p, __code next
81     (...)) { //:skip
82
83     goto switchvm_check_pgdirvm_impl(...);
84 }
85
86 __code init_inituvmvm_impl(struct vm_impl* vm, pde_t* pgdir, char* init,
87     uint sz, __code next(...)) {
88
89     Gearef(cbc_context, vm_impl)->pgdir = pgdir;
90     Gearef(cbc_context, vm_impl)->init = init;
91     Gearef(cbc_context, vm_impl)->sz = sz;
92     Gearef(cbc_context, vm_impl)->next = next;
93     goto init_inituvm_check_sz(vm, pgdir, init, sz, next(...));
94 }
95
96 __code loaduvmvm_impl(struct vm_impl* vm, pde_t* pgdir, char* addr,
97     struct inode* ip, uint offset, uint sz, __code next(...)) {
98     Gearef(cbc_context, vm_impl)->pgdir = pgdir;
99     Gearef(cbc_context, vm_impl)->addr = addr;
100    Gearef(cbc_context, vm_impl)->ip = ip;
101    Gearef(cbc_context, vm_impl)->offset = offset;
102    Gearef(cbc_context, vm_impl)->sz = sz;
103    Gearef(cbc_context, vm_impl)->next = next;
104
105    goto loaduvm_ptesize_checkvm_impl(vm, next(...));
106 }
107
108 __code allocuvmvm_impl(struct vm_impl* vm, pde_t* pgdir, uint oldsz, uint
109     newsz, __code next(...)) {
110
111     goto allocuvm_check_newszvm_impl(vm, pgdir, oldsz, newsz, next(...));
112 }
113
114 __code clearpteuvm_impl(struct vm_impl* vm, pde_t* pgdir, char* uva,
115     __code next(...)) {
116
117     goto clearpteu_check_ptevm_impl(vm, pgdir, uva, next(...));
118 }

```



```

115 | __code copyuvmvm_impl(struct vm_impl* vm, pde_t* pgdir, uint sz, __code
      |     next(...)) {
116 |
117 |     goto copyuvm_check_nullvm_impl(vm, pgdir, sz, __code next(...));
118 | }
119 |
120 | __code uva2kavm_impl(struct vm_impl* vm, pde_t* pgdir, char* uva, __code
      |     next(...)) {
121 |
122 |     goto uva2ka_check_pe_types(vm, pgdir, uva, next(...));
123 | }
124 |
125 | __code copyoutvm_impl(struct vm_impl* vm, pde_t* pgdir, uint va, void* pp
      |     , uint len, __code next(...)) {
126 |
127 |     vm->buf = (char*) pp;
128 |
129 |     goto copyout_loopvm_impl(vm, pgdir, va, pp, len, va0, pa0, next(...))
      |     ;
130 | }
131 |
132 | __code paging_intvm_impl(struct vm_impl* vm, uint phy_low, uint phy_hi,
      |     __code next(...)) {
133 |
134 |     goto paging_intvmvm_impl(vm, phy_low, phy_hi, next(...));
135 | }
136 |
137 | __code vm_void_ret(struct vm_impl* vm) {
138 |     return;
139 | }

```

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` の関数内で、インターフェースを `vm` で定義し、23行目の `vm->void_ret` のようにそれぞれの Code Gear 名に対応させていく。

CbCは1つ1つの Code Gear の信頼性を保障させるために細かくする必要があるので、`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`)。

### 5.3 インターフェース内の `private` メソッド

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 5.3 で定義する。

ソースコード 5.3: 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 loadvm_ptesize_check(Type* vm_impl, __code next(int ret, ...
33    );
34    __code loadvm_loop(Type* vm_impl, uint i, pte_t* pte, uint sz,
35    __code next(int ret, ...));
36    __code allocvm_check_newsz(Type* vm_impl, pde_t* pgdir, uint oldsz,
37    uint newsz, __code next(...));
38    __code allocvm_loop(Type* vm_impl, pde_t* pgdir, uint oldsz, uint
39    newsz, uint a, __code next(...));
40    __code copyvm_check_null(Type* vm_impl, pde_t* pgdir, uint sz,
41    __code next(...));
42    __code copyvm_loop(Type* vm_impl, pde_t* pgdir, uint sz, pde_t* d,
43    pte_t* pte, uint pa, uint i, uint ap, char* mem, __code next(int ret,
44    ...));
45    __code clearpteu_check_ptevm_impl(Type* vm_impl, pde_t* pgdir, char*
46    uva, __code next(...));
47    __code uva2ka_check_pe_types(Type* vm_impl, pde_t* pgdir, char* uva,
48    __code next(...));
49    __code paging_intvm_impl(Type* vm_impl, uint phy_low, uint phy_hi,
50    __code next(...));
51    __code copyout_loopvm_impl(Type* vm_impl, pde_t* pgdir, uint va, void
52    * pp, uint len, __code next(...));
53    __code switchvm_check_pgdirvm_impl(struct vm_impl* vm_impl, struct

```

```

proc* p, __code next(...));
43 __code init_initvm_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 if 文のある loaduvm という関数で説明を行う。

ソースコード 5.4: 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 (
9     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,
34             uint sz)
35 {
36     uint i, pa, n;
37     pte_t *pte;
38
39     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\_impl に記述する。(3 行目 11 行目)

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 ?? で示す。

ソースコード 5.5: private での loaduvm の実装

```

1 | #interface "vm_impl.h"
2 |
3 | __code loaduvm_ptesize_checkvm_impl(struct vm_impl* vm_impl, __code next(
4 |     int ret, ...)) {
5 |     char* addr = vm_impl->addr;
6 |
7 |     if ((uint) addr %PTE_SZ != 0) {
8 |         // goto panic
9 |     }
10 |
11 |     goto loaduvm_loopvm_impl(vm_impl, next(ret, ...));
12 | }
13 | __code loaduvm_loopvm_impl(struct vm_impl* vm_impl, __code next(int ret,
14 |     ...)) {
15 |     uint i = vm_impl->i;
16 |     uint sz = vm_impl->sz;
17 |
18 |     if (i < sz) {
19 |         goto loaduvm_check_pgdir(vm_impl, next(ret, ...));

```

```

19     }
20
21     goto loaduvm_exit(vm_impl, next(ret, ...));
22 }
23
24
25 static pte_t* walkpgdir (pde_t *pgdir, const void *va, int alloc)
26 {
27     pde_t *pde;
28     pte_t *pgtab;
29
30     // pgdir points to the page directory, get the page direcotry entry (
31     pde)
32     pde = &pgdir[PDE_IDX(va)];
33
34     if (*pde & PE_TYPES) {
35         pgtab = (pte_t*) p2v(PT_ADDR(*pde));
36     } else {
37         if (!alloc || (pgtab = (pte_t*) kpt_alloc()) == 0) {
38             return 0;
39         }
40
41         // Make sure all those PTE_P bits are zero.
42         memset(pgtab, 0, PT_SZ);
43
44         // The permissions here are overly generous, but they can
45         // be further restricted by the permissions in the page table
46         // entries, if necessary.
47         *pde = v2p(pgtab) | UPDE_TYPE;
48     }
49
50     return &pgtab[PTE_IDX(va)];
51 }
52
53
54 __code loaduvm_check_pgdir(struct vm_impl* vm_impl, __code next(int ret,
55 ...) {
56     pte_t* pte = vm_impl->pte;
57     pde_t* pgdir = vm_impl->pgdir;
58     uint i = vm_impl->i;
59     char* addr = vm_impl->addr;
60     uint pa = vm_impl->pa;
61
62     if ((pte = walkpgdir(pgdir, addr + i, 0)) == 0) {
63         // goto panic
64     }
65     pa = PTE_ADDR(*pte);
66
67     vm_impl->pte = pte;
68     vm_impl->pgdir = pgdir;
69     vm_impl->addr = addr;
70     vm_impl->pa = pa;

```

```

71 |     goto loaduvm_check_PTE_SZ(vm_impl, next(ret, ...));
72 | }
73 |
74 | __code loaduvm_check_PTE_SZ(struct vm_impl* vm_impl, __code next(int ret,
75 |     ...)) {
76 |     uint sz = vm_impl->sz;
77 |     uint i = vm_impl->i;
78 |     uint n = vm_impl->n;
79 |     struct inode* ip = vm_impl->ip;
80 |     uint pa = vm_impl->pa;
81 |     uint offset = vm_impl->offset;
82 |
83 |     if (sz - i < PTE_SZ) {
84 |         n = sz - i;
85 |     } else {
86 |         n = PTE_SZ;
87 |     }
88 |
89 |     if (readi(ip, p2v(pa), offset + i, n) != n) {
90 |         ret = -1;
91 |         goto next(ret, ...);
92 |     }
93 |
94 |     vm_impl->n = n;
95 |
96 |     goto loaduvm_loopvm_impl(vm_impl, next(ret, ...));
97 | }
98 | __code loaduvm_exit(struct vm_impl* vm_impl, __code next(int ret, ...)) {
99 |     ret = 0;
100 |     goto next(ret, ...);
101 | }

```

for 文から末尾再起の変換について  
static なものに関しては後々実装する

## 5.4 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。例として、ソースコード 5.6 の 16 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 17 行目以降が実行されなくなる。

ソースコード 5.6: 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,
16 |   _binary_initcode_start, (int)_binary_initcode_size);
17 |   p->sz = PTE_SZ;
18 |
19 |   // craft the trapframe as if
   memset(p->tf, 0, sizeof(*p->tf));

```

ソースコード 5.7: dummy を使った呼び出し

```

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

```

```
32 |  
33 |     if((p->pgdir = kpt_alloc()) == NULL) {  
34 |         panic("userinit: out of memory?");  
35 |     }  
36 |  
37 |     dummy(p, _binary_initcode_start, _binary_initcode_size);
```

ソースコードの説明

- dummy を使って解決する



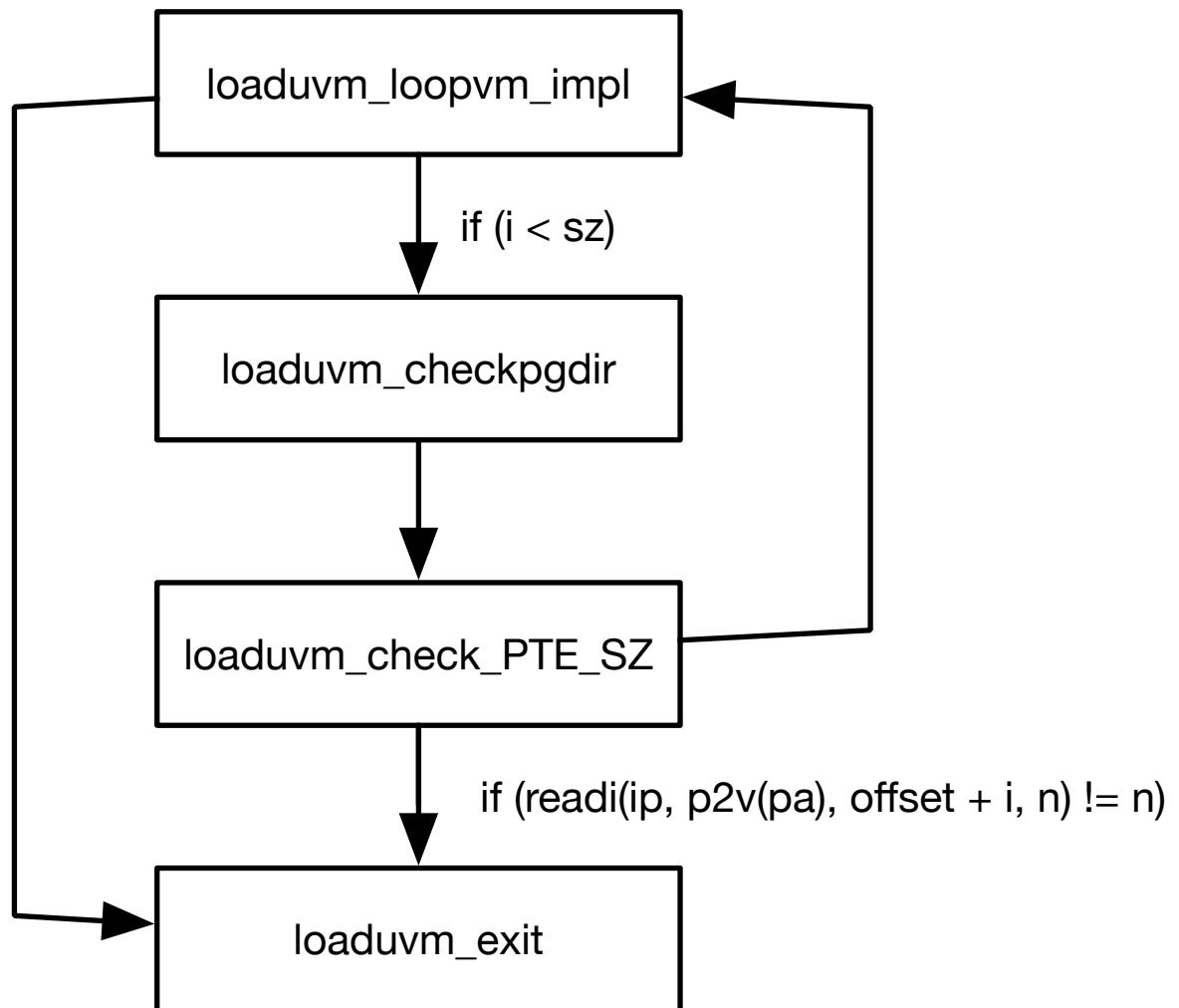


図 5.1: CbC によるループの実装

## 第6章 評価

## 第7章 まとめ

### 7.1 今後の書き換え方針

本論文では、Paging 部分の書き換えの説明を行なった。今後 Xv6 全体の CbC による書き換えを行なっていく。

# 謝辞

2020年3月  
桃原 優

## 参考文献

- [1] Russ Cox, M. Frans Kaashoek, and Robert T. Morris. xv6: a simple, Unix-like teaching operating system. 2014.
- [2] Herbert Bos and Andrew S. Tanenbaum. Modern Operating Systems. 2015.
- [3] Raspberry Pi — Teach, Learn, and Make with Raspberry Pi. <https://www.raspberrypi.org>.
- [4] Zhiyi Wang. xv6-rpi. <https://code.google.com/archive/p/xv6-rpi/>, 2013.
- [5] Kaito TOKKMORI and Shinji KONO. Implementing continuation based language in llvm and clang. *LOLA 2015*, July 2015.
- [6] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, Vol. 93, No. 1, pp. 55–92, July 1991.
- [7] Jean Yang and Chris Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pp. 99–110, New York, NY, USA, 2010. ACM.
- [8] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pp. 207–220, New York, NY, USA, 2009. ACM.
- [9] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pp. 1–16, Berkeley, CA, USA, 2016. USENIX Association.

- [10] 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.
- [11] 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.
- [12] 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.
- [13] GNU Compiler Collection (GCC) Internals. <http://gcc.gnu.org/onlinedocs/gccint/>.
- [14] 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.
- [15] ARM Architecture Reference Manual. <http://infocenter.arm.com/help/topic/com.arm.doc.subset.architecture.reference/index.html>.

## 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

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

### A-1 インターフェース内の `private` メソッドの実装

Xv6 の元のコードである `vm.c` A.1 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.2 に示す。

ソースコード A.1: 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 initial 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 (
95 |     pde)
96 |     pde = &pgdir[PDE_IDX(va)];
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 switchvmm (struct proc *p)
163 {
164     uint val;
165
166     pushcli();
167
168     if (p->pgdir == 0) {
169         panic("switchvmm: 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 initvmm (pde_t *pgdir, char *init, uint sz)
182 {
183     char *mem;
184
185     if (sz >= PTE_SZ) {
186         panic("initvmm: 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,
198 |             uint sz)
199 | {
200 |     uint i, pa, n;
201 |     pte_t *pte;
202 |
203 |     if ((uint) addr % PTE_SZ != 0) {
204 |         panic("loaduvm: addr must be page aligned");
205 |     }
206 |
207 |     for (i = 0; i < sz; i += PTE_SZ) {
208 |         if ((pte = walkpgdir(pgdir, addr + i, 0)) == 0) {
209 |             panic("loaduvm: address should exist");
210 |         }
211 |
212 |         pa = PTE_ADDR(*pte);
213 |
214 |         if (sz - i < PTE_SZ) {
215 |             n = sz - i;
216 |         } else {
217 |             n = PTE_SZ;
218 |         }
219 |
220 |         if (readi(ip, p2v(pa), offset + i, n) != n) {
221 |             return -1;
222 |         }
223 |     }
224 |
225 |     return 0;
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 |
230 | int allocuvm (pde_t *pgdir, uint oldsz, uint newsz)
231 | {
232 |     char *mem;
233 |     uint a;
234 |
235 |     if (newsz >= UADDR_SZ) {
236 |         return 0;
237 |     }
238 |
239 |     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,
446 |              phy_low, AP_KU);
447 |     flush_tlb ();
447 | }
```



## ソースコード A.2: 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 loadvm_ptesize_checkvm_impl(struct vm_impl* vm_impl, __code next(
13     int ret, ...)) {
14     char* addr = vm_impl->addr;
15
16     if ((uint) addr %PTE_SZ != 0) {
17         // goto panic
18     }
19
20     goto loadvm_loopvm_impl(vm_impl, next(ret, ...));
21 }
22
23 __code loadvm_loopvm_impl(struct vm_impl* vm_impl, __code next(int ret,
24     ...)) {
25     uint i = vm_impl->i;
26     uint sz = vm_impl->sz;
27
28     if (i < sz) {
29         goto loadvm_check_pgdir(vm_impl, next(ret, ...));
30     }
31
32     goto loadvm_exit(vm_impl, next(ret, ...));
33 }
34
35 static pte_t* walkpgdir (pde_t *pgdir, const void *va, int alloc)
36 {
37     pde_t *pde;
38     pte_t *pgtab;
39
40     // pgdir points to the page directory, get the page direcotry entry (
41     pde)
42     pde = &pgdir[PDE_IDX(va)];
43
44     if (*pde & PE_TYPES) {
45         pgtab = (pte_t*) p2v(PT_ADDR(*pde));
46     } else {
47         if (!alloc || (pgtab = (pte_t*) kpt_alloc()) == 0) {
48             return 0;
49         }
50     }
51 }

```

```

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 loadvm_check_pgdir(struct vm_impl* vm_impl, __code next(int ret,
64 |     ...)) {
65 |     pte_t* pte = vm_impl->pte;
66 |     pde_t* pgdir = vm_impl->pgdir;
67 |     uint i = vm_impl->i;
68 |     char* addr = vm_impl->addr;
69 |     uint pa = vm_impl->pa;
70 |
71 |     if ((pte = walkpgdir(pgdir, addr + i, 0)) == 0) {
72 |         // goto panic
73 |     }
74 |     pa = PTE_ADDR(*pte);
75 |
76 |     vm_impl->pte = pte;
77 |     vm_impl->pgdir = pgdir;
78 |     vm_impl->addr = addr;
79 |     vm_impl->pa = pa;
80 |
81 |     goto loadvm_check_PTE_SZ(vm_impl, next(ret, ...));
82 | }
83 | __code loadvm_check_PTE_SZ(struct vm_impl* vm_impl, __code next(int ret,
84 |     ...)) {
85 |     uint sz = vm_impl->sz;
86 |     uint i = vm_impl->i;
87 |     uint n = vm_impl->n;
88 |     struct inode* ip = vm_impl->ip;
89 |     uint pa = vm_impl->pa;
90 |     uint offset = vm_impl->offset;
91 |
92 |     if (sz - i < PTE_SZ) {
93 |         n = sz - i;
94 |     } else {
95 |         n = PTE_SZ;
96 |     }
97 |
98 |     if (readi(ip, p2v(pa), offset + i, n) != n) {
99 |         ret = -1;
100 |         goto next(ret, ...);
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 allocvm_check_newszvm_impl(struct vm_impl* vm_impl, pde_t* pgdir,
169 |     uint oldsz, uint newsz, __code next(int ret, ...)){
170 |     if (newsz >= UADDR_SZ) {
171 |         goto next(0, ...);
172 |     }
173 |
174 |     if (newsz < oldsz) {
175 |         ret = newsz;
176 |         goto next(ret, ...);
177 |     }
178 |
179 |     char* mem;
180 |     uint a = align_up(oldsz, PTE_SZ);
181 |
182 |     goto allocvm_loopvm_impl(vm_impl, pgdir, oldsz, newsz, mem, a, next(
183 |     ret, ...));
184 | }
185 |
186 | __code allocvm_loopvm_impl(struct vm_impl* vm_impl, pde_t* pgdir, uint
187 |     oldsz, uint newsz, char* mem, uint a, __code next(int ret, ...)){
188 |
189 |     if (a < newsz) {
190 |         mem = alloc_page();
191 |
192 |         if (mem == 0) {
193 |             cprintf("allocvm out of memory\n");
194 |             deallocvm(pgdir, newsz, oldsz);
195 |             goto next(0, ...);
196 |         }
197 |
198 |         memset(mem, 0, PTE_SZ);
199 |         mappages(pgdir, (char*) a, PTE_SZ, v2p(mem), AP_KU);
200 |
201 |         goto allocvm_loopvm_impl(vm_impl, pgdir, oldsz, newsz, a +
202 |         PTE_SZ, next(ret, ...));
203 |     }
204 |     ret = newsz;
205 |     goto next(ret, ...);
206 | }
207 |
208 | __code clearpteu_check_ptevm_impl(struct vm_impl* vm_impl, pde_t* pgdir,

```

```

205 char* uva, __code next(int ret, ...)) {
206     pte_t *pte;
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_K0 << 4;
215
216     goto next(ret, ...);
217 }
218
219 __code copyvm_check_nullvm_impl(struct vm_impl* vm_impl, pde_t* pgdir,
220     uint sz, __code next(int ret, ...)) {
221     pde_t *d;
222     pte_t *pte;
223     uint pa, i, ap;
224     char *mem;
225
226     // allocate a new first level page directory
227     d = kpt_alloc();
228     if (d == NULL) {
229         ret = NULL;
230         goto next(ret, ...);
231     }
232     i = 0;
233     goto copyvm_loopvm_impl(vm_impl, pgdir, sz, *d, *pte, pa, i, ap, *
234     mem, next(ret, ...));
235 }
236 __code copyvm_loopvm_impl(struct vm_impl* vm_impl, pde_t* pgdir, uint sz
237     , pde_t* d, pte_t* pte, uint pa, uint i, uint ap, char* mem, __code
238     next(int ret, ...)) {
239     if (i < sz) {
240         goto copyvm_loop_check_walkpgdir(vm_impl, pgdir, sz, d, pte, pa,
241         i, ap, mem, __code next(int ret, ...));
242     }
243     ret = d;
244     goto next(ret, ...);
245 }
246 __code copyvm_loop_check_walkpgdir(struct vm_impl* vm_impl, pde_t* pgdir
247     , uint sz, pde_t* d, pte_t* pte, uint pa, uint i, uint ap, char* mem,
248     __code next(int ret, ...)) {
249     if ((pte = walkpgdir(pgdir, (void *) i, 0)) == 0) {
250         // panic("copyvm: pte should exist");
251         // goto panic();

```

```

250     }
251     goto copyuvm_loop_check_pte(vm_impl, pgdir, sz, d, pte, pa, i, ap,
mem, __code next(int ret, ...));
252 }
253
254 __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, ...)) {
255     if (!(*pte & PE_TYPES)) {
256         // panic("copyuvm: page not present");
257         // goto panic();
258     }
259
260     goto copyuvm_loop_check_mem(vm_impl, pgdir, sz, d, pte, pa, i, ap,
mem, __code next(int ret, ...));
262 }
263
264 __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, ...)) {
265     pa = PTE_ADDR (*pte);
266     ap = PTE_AP (*pte);
267
268     if ((mem = alloc_page()) == 0) {
269         goto copyuvm_loop_bad(vm_impl, d, next(...));
270     }
271     goto copyuvm_loop_check_mappages(vm_impl, pgdir, sz, d, pte, pa, i,
ap, mem, __code next(int ret, ...));
272 }
273
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 switchvm_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("switchvm: 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 | }
```