修士 (工学) 学位論文
Master's Thesis of Engineering

GearsOS の Paging

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

時代とともに進歩するハードウェアやソフトウェアに対して、OS 自体に信頼性が求められる. メモリ管理は OS の信頼性の基本であるが、 現代の OS では、 User Space で Page Table Entry によるメモリ管理を行える OS は少ない、 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 の信頼性の基本であるメモリ管理を行える OS を実装することで、 OS の信頼性を保証したい. 既存の Gears OS でのメモリ管理では単に Page Table Entry をコピーする Fork を実装しているが、 さらに資源管理を行える CbC で軽量なハードウェアでも動かせるように Arm のバイナリを 出力する Xv6 を参考に GearsOS にメモリ管理を行う API を考察する。

Abstract

目 次

図 目 次

表 目 次

ソースコード目次

第1章　メモリ管理による信頼性の保証

メモリ管理は OS の信頼性の基本であるが、 現代の OS では、 User Space で Page Table Entry によるメモリ管理を行える OS は少ない。これは User レベルの操作で Page Table が書き換えられたり、別の Page にアクセスするのを防ぐためだと考えられる。しかし、User Space でメモリ管理を行えるようにすることで、 Page のバリデーションをチェックしたり、サンドボックスによる信頼性の保証を行えるようになる。また、適切な記述をすれば最適なメモリ管理にも繋がる。

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 の信頼性の基本であるメモリ管理を行える OS を実装することで、 OS の信頼性を保証したい。既存の Gears OS でのメモリ管理では単に Page Table Entry をコピーする Fork を実装しているが、 さらに資源管理を行える CbC で軽量なハードウェアでも動かせるように Arm のバイナリを出力する Xv6 を参考に GearsOS にメモリ管理を行う API を考察する。

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。CbC で書き換えることによって OS の機能の信頼性を保証することができる。

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。j Code Gear 間の処理の流れを図 2.1 に示す。

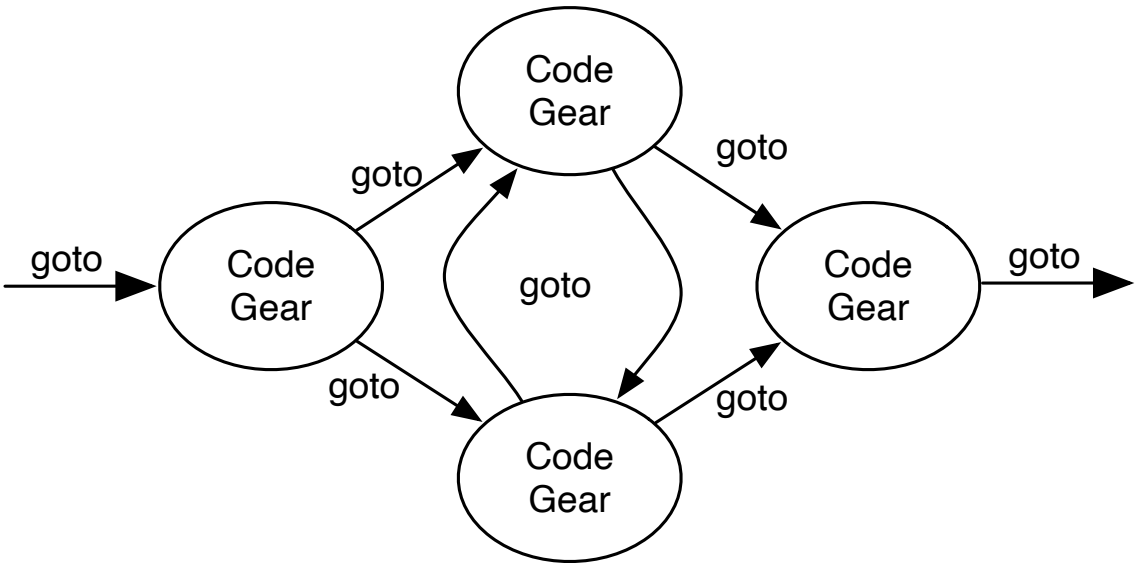

図 2.1: Code Gear 間の継続

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。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。

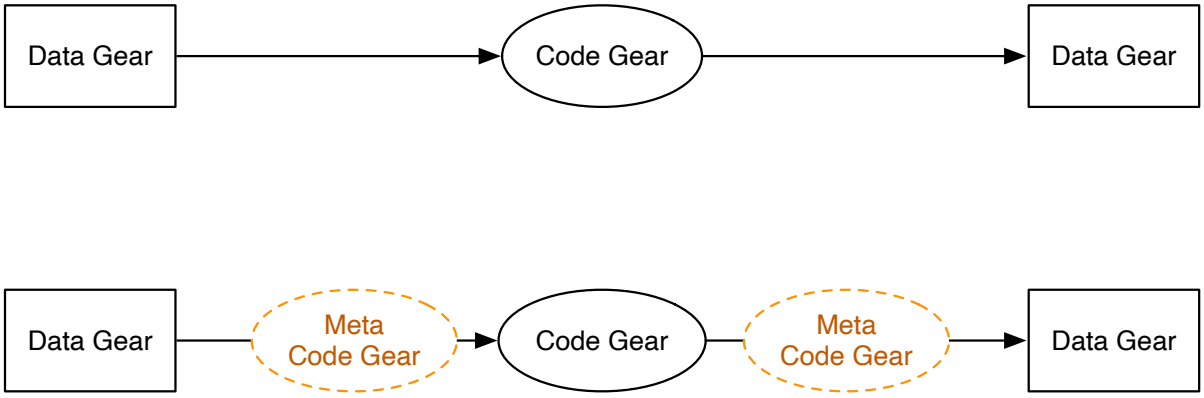

図 2.2: ノーマルレベルとメタレベルの継続の見え方

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。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 Gaer 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。

```
1  /* Context definition */
2  struct Context {
3      enum Code next;
4      int codeNum;
5      __code (**code) (struct Context*);
6      union Data **data;
7      void* heapStart;
8      void* heap;
9      long heapLimit;
10     int dataNum;
11
12     // task parameter
13     int idgCount; //number of waiting dataGear
14     int idg;
15     int maxIdg;
16     int odg;
17     int maxOdg;
18     int gpu; // GPU task
19     struct Worker* worker;
20     struct TaskManager* taskManager;
21     struct Context* task;
22     struct Element* taskList;
23 #ifdef USE_CUDAWorker
24     int num_exec;
25     CUmodule module;
26     CUfunction function;
27 #endif
28     /* multi dimension parameter */
29     int iterate;
30     struct Iterator* iterator;
31 };
32
33 union Data {
34     struct Meta {
35         enum DataType type;
36         long size;
37         long len;
38         struct Queue* wait; // tasks waiting this dataGear
39     } Meta;
40     struct Context Context;
41     // Stack Interface
42     struct Stack {
43         union Data* stack;
```

```
44          union Data* data;
45          union Data* data1;
46          enum Code whenEmpty;
47          enum Code clear;
48          enum Code push;
49          enum Code pop;
50          enum Code pop2;
51          enum Code isEmpty;
52          enum Code get;
53          enum Code get2;
54          enum Code next;
55      } Stack;
56      // Stack implementations
57      struct SingleLinkedStack {
58          struct Element* top;
59      } SingleLinkedStack;
60      ....
61  }; // union Data end       this is necessary for context generator
```

ソースコードの説明書く

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。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。

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 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

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

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。

図 ?? で Xv6 の仮想メモリと実メモリについて説明する。

4.3　User Space で Paging をする利点

Context に必要な Page Tbale を提供する Interface と User Space からアクセスする API 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。 Meta Computation レベルで処理することで Use Space でも Page Table を操作することができる。Meta Computation に戻る際に、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。

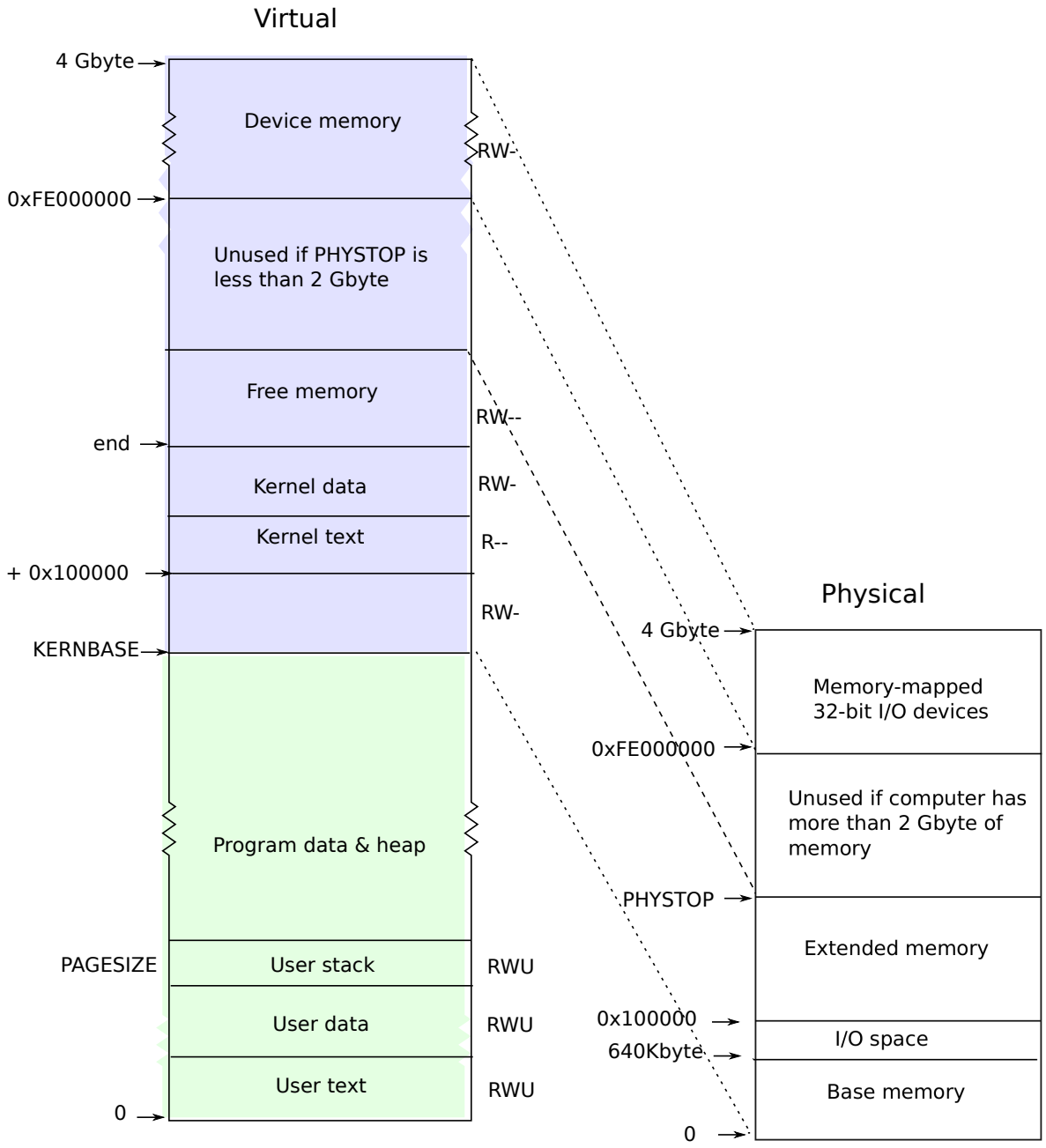

図 4.1: Layout of the virtual address space of a process and the layout of the physical address space. Note that if a machine has more than 2 Gbyte of physical memory, xv6 can use only the memory that fits between KERNBASE and 0xFE00000. Russ Cox(2018) xv6 a simple, Unix-like teaching operating system (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 5.1 に示す。

ソースコード 5.1: vm のインターフェース

```
1  typedef struct vm<Type,Impl> {
2      union Data* vm;
3      uint low;
4      uint hi;
5      struct proc* p;
6      pde_t* pgdir;
7      char* init;
8      uint sz;
9      char* addr;
10     struct inode* ip;
11     uint offset;
12     uint oldsz;
13     uint newsz;
14     char* uva;
```

```
15      uint va;
16      void* pp;
17      uint len;
18      uint phy_low;
19      uint phy_hi;
20      __code init_vmm(Impl* vm, __code next(...));
21      __code kpt_freerange(Impl* vm, uint low, uint hi, __code next(...));
22      __code kpt_alloc(Impl* vm ,__code next(...));
23      __code switchuvm(Impl* vm ,struct proc* p, __code next(...));
24      __code init_inituvm(Impl* vm, pde_t* pgdir, char* init, uint sz,
        __code next(...));
25      __code loaduvm(Impl* vm,pde_t* pgdir, char* addr, struct inode* ip,
        uint offset, uint sz,  __code next(...));
26      __code allocuvm(Impl* vm, pde_t* pgdir, uint oldsz, uint newsz,
        __code next(...));
27      __code clearpteu(Impl* vm, pde_t* pgdir, char* uva,  __code next(...)
        );
28      __code copyuvm(Impl* vm, pde_t* pgdir, uint sz, __code next(...));
29      __code uva2ka(Impl* vm, pde_t* pgdir, char* uva, __code next(...));
30      __code copyout(Impl* vm, pde_t* pgdir, uint va, void* pp, uint len,
        __code next(...));
31      __code paging_int(Impl* vm, uint phy_low, uint phy_hi, __code next
        (...));
32      __code void_ret(Impl* vm);
33      __code next(...);
34  } vm;
```

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 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 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。

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。

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 switchuvmvm_impl(struct vm_impl* vm , struct proc* p, __code next
       (...)) { //:skip
81
82      goto switchuvm_check_pgdirvm_impl(...);
83  }
84
85  __code init_inituvmvm_impl(struct vm_impl* vm, pde_t* pgdir, char* init,
       uint sz, __code next(...)) {
86
87      Gearef(cbc_context, vm_impl)->pgdir = pgdir;
88      Gearef(cbc_context, vm_impl)->init = init;
89      Gearef(cbc_context, vm_impl)->sz = sz;
90      Gearef(cbc_context, vm_impl)->next = next;
91      goto init_inituvm_check_sz(vm, pgdir, init, sz, next(...));
92  }
93
94  __code loaduvmvm_impl(struct vm_impl* vm, pde_t* pgdir, char* addr,
       struct inode* ip, uint offset, uint sz,  __code next(...)) {
95      Gearef(cbc_context, vm_impl)->pgdir = pgdir;
96      Gearef(cbc_context, vm_impl)->addr = addr;
97      Gearef(cbc_context, vm_impl)->ip = ip;
98      Gearef(cbc_context, vm_impl)->offset = offset;
99      Gearef(cbc_context, vm_impl)->sz = sz;
100     Gearef(cbc_context, vm_impl)->next = next;
101
102     goto loaduvm_ptesize_checkvm_impl(vm, next(...));
103 }
104
105 __code allocuvmvm_impl(struct vm_impl* vm, pde_t* pgdir, uint oldsz, uint
        newsz, __code next(...)) {
106
```

```
107     goto allocuvm_check_newszvm_impl(vm, pgdir, oldsz, newsz, next(...));
108 }
109
110 __code clearpteuvm_impl(struct vm_impl* vm, pde_t* pgdir, char* uva,
        __code next(...)) {
111
112     goto clearpteu_check_ptevm_impl(vm, pgdir, uva, next(...));
113 }
114
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 }
```

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 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
2  typedef struct vm_impl<Impl, Isa> impl vm{
3      union Data* vm_impl;
4      uint i;
5      pte_t* pte;
6      uint sz;
7      pde_t* pgdir;
8      char* addr;
9      struct inode* ip;
10     uint offset;
11     uint pa;
12     uint n;
13     uint oldsz;
14     uint newsz;
15     uint a;
16     int ret;
17     char* mem;
18     char* uva;
19     pde_t* d;
20     uint ap;
21     uint phy_low;
22     uint phy_hi;
23     uint va;
24     void* pp;
25     uint len;
26     char* buf;
27     char* pa0;
28     uint va0;
29     proc_struct* p;
30     char* init;
31
32     __code kpt_alloc_check_impl(Type* vm_impl, __code next(...));
33     __code loaduvm_ptesize_check(Type* vm_impl, __code next(int ret, ...)
       );
34     __code loaduvm_loop(Type* vm_impl, uint i, pte_t* pte, uint sz,
       __code next(int ret, ...));
35     __code allocuvm_check_newsz(Type* vm_impl, pde_t* pgdir, uint oldsz,
       uint newsz, __code next(...));
36     __code allocuvm_loop(Type* vm_impl, pde_t* pgdir, uint oldsz, uint
       newsz, uint a, __code next(...));
37     __code copyuvm_check_null(Type* vm_impl, pde_t* pgdir, uint sz,
       __code next(...));
38     __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,
       ...));
39     __code clearpteu_check_ptevm_impl(Type* vm_impl, pde_t* pgdir, char*
       uva,  __code next(...));
40     __code uva2ka_check_pe_types(Type* vm_impl, pde_t* pgdir, char* uva,
       __code next(...));
41     __code paging_intvm_impl(Type* vm_impl, uint phy_low, uint phy_hi,
       __code next(...));
42     __code copyout_loopvm_impl(Type* vm_impl, pde_t* pgdir, uint va, void
       * pp, uint len, __code next(...));
43     __code switchuvm_check_pgdirvm_impl(struct vm_impl* vm_impl, struct
```

```
    proc* p, __code next(...));
44   __code init_inituvm_check_sz(struct vm_impl* vm_impl, pde_t* pgdir,
    char* init, uint sz, __code next(...));
45   __code void_ret(Type* vm_impl);
46   __code next(...);
47 } vm_impl;
```

インターフェースを vm で インターフェースの実装を 5.4 で示す。

ソースコード 5.4: vm private 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
39 extern struct {
40     struct spinlock lock;
41     struct run *freelist;
42 } kpt_mem;
43
```

```
44 __code init_vmmvm_impl(struct vm_impl* vm,__code next(...)) {
45     initlock(&kpt_mem.lock, "vm");
46     kpt_mem.freelist = NULL;
47 
48     goto next(...);
49 }
50 
51 extern struct run {
52     struct run *next;
53 };
54 
55 static void _kpt_free (char *v)
56 {
57     struct run *r;
58 
59     r = (struct run*) v;
60     r->next = kpt_mem.freelist;
61     kpt_mem.freelist = r;
62 }
63 
64 __code kpt_freerangevm_impl(struct vm_impl* vm, uint low, uint hi, __code
       next(...)) {
65 
66    if (low < hi) {
67      _kpt_free((char*)low);
68      goto kpt_freerangevm_impl(vm, low + PT_SZ, hi, next(...));
69 
70    }
71   goto next(...);
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 switchuvmvm_impl(struct vm_impl* vm , struct proc* p, __code next
      (...)) { //:skip
81 
82     goto switchuvm_check_pgdirvm_impl(...);
83 }
84 
85 __code init_inituvmvm_impl(struct vm_impl* vm, pde_t* pgdir, char* init,
      uint sz, __code next(...)) {
86 
87     Gearef(cbc_context, vm_impl)->pgdir = pgdir;
88     Gearef(cbc_context, vm_impl)->init = init;
89     Gearef(cbc_context, vm_impl)->sz = sz;
90     Gearef(cbc_context, vm_impl)->next = next;
91     goto init_inituvm_check_sz(vm, pgdir, init, sz, next(...));
92 }
93 
```

```
94  __code loaduvmvm_impl(struct vm_impl* vm, pde_t* pgdir, char* addr,
        struct inode* ip, uint offset, uint sz,  __code next(...)) {
95      Gearef(cbc_context, vm_impl)->pgdir = pgdir;
96      Gearef(cbc_context, vm_impl)->addr = addr;
97      Gearef(cbc_context, vm_impl)->ip = ip;
98      Gearef(cbc_context, vm_impl)->offset = offset;
99      Gearef(cbc_context, vm_impl)->sz = sz;
100     Gearef(cbc_context, vm_impl)->next = next;
101
102     goto loaduvm_ptesize_checkvm_impl(vm, next(...));
103 }
104
105 __code allocuvmvm_impl(struct vm_impl* vm, pde_t* pgdir, uint oldsz, uint
        newsz, __code next(...)) {
106
107     goto allocuvm_check_newszvm_impl(vm, pgdir, oldsz, newsz, next(...));
108 }
109
110 __code clearpteuvm_impl(struct vm_impl* vm, pde_t* pgdir, char* uva,
        __code next(...)) {
111
112     goto clearpteu_check_ptevm_impl(vm, pgdir, uva, next(...));
113 }
114
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 }
```

5.4 インターフェースの呼び出し

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、5.5 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 5.5: cbc インターフェースの goto

```
1
2  void userinit(void)
3  {
4      struct proc* p;
5      extern char _binary_initcode_start[], _binary_initcode_size[];
6
7      p = allocproc();
8      initContext(&p->cbc_context);
9
10     initproc = p;
11
12     if((p->pgdir = kpt_alloc()) == NULL) {
13         panic("userinit: out of memory?");
14     }
15
16     goto cbc_init_vmm_dummy(&p->cbc_context, p, p->pgdir,
       _binary_initcode_start, (int)_binary_initcode_size);
17     p->sz = PTE_SZ;
18
19     // craft the trapframe as if
20     memset(p->tf, 0, sizeof(*p->tf));
```

ソースコード 5.6: dummy を使った呼び出し

```
1
2
3  void dummy(struct proc *p, char _binary_initcode_start[], char
       _binary_initcode_size[])
4  {
5      // inituvm(p->pgdir, _binary_initcode_start, (int)
       _binary_initcode_size);
6      goto cbc_init_vmm_dummy(&p->cbc_context, p, p->pgdir,
       _binary_initcode_start, (int)_binary_initcode_size);
7
8  }
9
10
11
12 __ncode cbc_init_vmm_dummy(struct Context* cbc_context, struct proc* p,
       pde_t* pgdir, char* init, uint sz){//:skip
```

```
13
14      struct vm* vm = createvm_impl(cbc_context);
15      // goto vm->init_vmm(vm, pgdir, init, sz , vm->void_ret);
16          Gearef(cbc_context, vm)->vm = (union Data*) vm;
17          Gearef(cbc_context, vm)->pgdir = pgdir;
18          Gearef(cbc_context, vm)->init = init;
19          Gearef(cbc_context, vm)->sz = sz ;
20          Gearef(cbc_context, vm)->next = C_vm_void_ret ;
21      goto meta(cbc_context, vm->init_inituvm);
22  }
23
24
25  void userinit(void)
26  {
27      struct proc* p;
28      extern char _binary_initcode_start[], _binary_initcode_size[];
29
30      p = allocproc();
31      initContext(&p->cbc_context);
32
33      initproc = p;
34
35      if((p->pgdir = kpt_alloc()) == NULL) {
36          panic("userinit: out of memory?");
37      }
38
39      dummy(p, _binary_initcode_start, _binary_initcode_size);
```

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 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*.