CbC インターフェースによる CbCXv6 の書き換え

2020/2/18

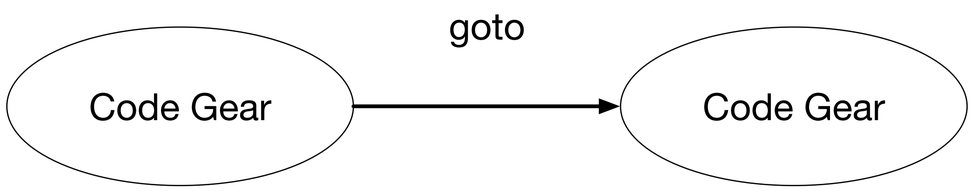
桃原 優　並列信頼研

OSの信頼性の保証

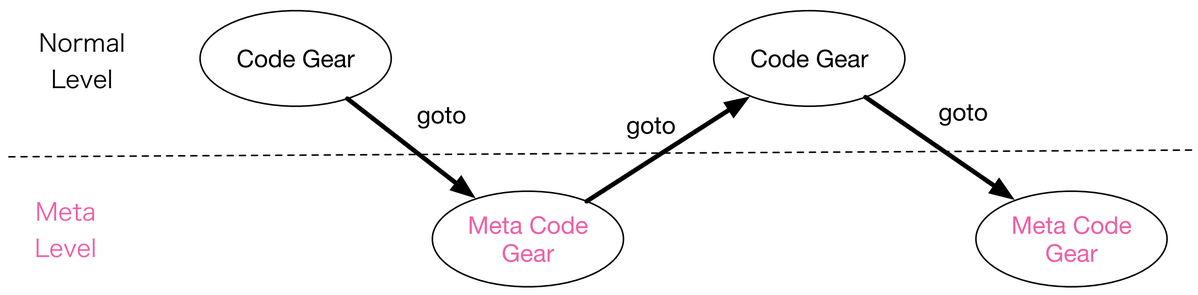
・OSに信頼性が求められるが、複雑な機能が多く、**全てのコードに対して検証を行うことは困難**である
・実際にパスワードなしで root にアクセスできてしまうという重大なバグも発生した
・信頼性を保証するのに適した CbC を使って OS を書き換える
・CbC は状態遷移ベースで記述され、入力に対する出力を検証する事で信頼性を保証する

Continuation based C (CbC)

・本研究室で開発しているプログラミング言語
・状態遷移ベースで記述するため、**検証しやすい**
・**基本的な処理の単位である Code Gear** で記述し、goto によって遷移する
・**Code Gear からアクセスできるデータの単位を Data Gear** と呼ぶ
・Code Gear に入力されるデータを Input Data Gear、出力されるデータを Output Data Gear と呼ぶ



・通常記述できないメモリなどの資源管理部分を**メタレベル**と呼ぶ
・CbC はメタレベルも記述することができる
・メタレベルにも処理の単位である Meta Code Gear と データの単位である Meta Data Gear が存在する
・メタレベルで見ると **Code Gear の間に Meta Code Gear が挟まっている**
・Meta Data Gear はノーマルレベルで資源管理部分の書き換えやアクセスを防ぐために存在する



Context

・Meta Data Gear の1つ
・全ての Data Gear と Code Gear が登録されている
・xv6 では1つのユーザープロセスに対して1つの Context が存在する
・**Context を切り替えて実行環境を入れ替える**事によって VM やコンテナの実装を目指している

xv6

・マサチューセッツ工科大の講義用教材として作られた OS
・1万行程ので書かれている計量な OS
・カーネルを採用していて、ユーザープログラムとは分離されている
　・重要なファイルへのアクセスや書き込みを防ぐため
・カーネルは以下をユーザープログラムに提供している
　・プロセス管理、メモリ管理、ファイルシルテム等
・ARM のバイナリを出力できるので Raspberry Pi でも動かせる
・xv6 を CbC で書き換える事で OS を実装する

Paging

・OS の信頼性を保証する上で重要なメモリ管理部分から書き直していく。xv6 ではメモリ管理には Paging という手法が用いられている。
・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
・xv6 内で Paging を行う vm.c で主要な関数を説明する
　・**init_vmm**：最初に外から呼び出される関数
　・**loaduvm**：スワップ領域から呼び出される。カーネル内をループしてユーザープロセス側が待ち状態にする
　・**kpt_free**：カーネルのメモリ解放
　・**kpt_alloc**：カーネルのメモリ割り当て
　・**initumv**：Page の作成
　・**switchuvm**：Page の切り替え

インターフェースの定義(vm.h)

・メモリ管理部分である vm.c のインターフェースの書き換えについて説明する。インターフェースの定義は vm.h で行う

```
typedef struct vm<Type,Impl> {
    __code init_vmm(Impl* vm, __code next(...));
    __code loaduvm(Impl* vm,pde_t* pgdir, char* addr, struct
inode* ip, uint offset, uint sz,  __code next(...));
```

・1行目の typedef struct のあとにインターフェース名(vm)を書く
・2行目以降で vm で使う Code Gear を __code CodeGear名() で登録する。引数が Data Gear に相当する

インターフェースの実装(vm_impl.cbc)

・定義をしたインターフェースを使用するために初期化を行う
・メモリ上にインターフェースの置き場所と実装(vm_impl)を確保

```
#interface "vm.h"
vm* createvm_impl(struct Context* cbc_context) {
    struct vm* vm  = new vm();
    struct vm_impl* vm_impl = new vm_impl();
    vm->vm = (union Data*)vm_impl;
// …
    vm->loaduvm = C_loaduvmvm_impl;
```

・2行目の typedef struct のあとにインターフェース名(vm)を書く
・3行目以降で vm で使う Code Gear を __code CodeGear名() で登録する。引数が Data Gear に相当する
・最後の行でインターフェースと実装の番号を紐付けている
・最後にインターフェースの使用例を示す

```
__code loaduvmvm_impl(struct vm_impl* vm, pde_t* pgdir, char*
addr, struct inode* ip, uint offset, uint sz,  __code
next(...)) {
    goto loaduvm_ptesize_checkvm_impl(vm, next(...));
}
```

・Code Gear内に処理を書き、実装を分ける場合は goto で遷移する

まとめと今後の課題

・OS 内部で CbC インターフェースを扱えるようになった
・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
・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