Ramooflax

pre-boot virtualization

Stephane Duverger

Innovation Works
Suresnes, France

Tokyo, Nov. 2011

はじめに

こんなツールが欲しかった ...

- 複雑なシステムの制御機能を持つ（BIOS、カーネルなど ...）
- 単一の**物理**マシン上で動作（32ビット・64ビットのx86 マシン）
- ソフトウェアに依存しない

こんなツールが欲しかった ...

- 複雑なシステムの制御機能を持つ（BIOS、カーネルなど ...）
- 単一の**物理**マシン上で動作（32ビット・64ビットのx86 マシン）
- ソフトウェアに依存しない

アイデア

- 単一のVMのみが動作するハイパーバイザー（VMM）
- リモート制御可能
- Type 1（ベアメタル）のハイパーバイザー
  - シンプルな隔離方式
  - ハードウェアを明確にコントロールできる
  - ソフトウェアに依存しない！
  - VMの前にスタートアップ・リクエストを発行

既存のハイパーバイザーについて

一般的な解決策

- VirtualBox、KVMなど：type 2（ホスト）のため不適切
- Xen: 導入・展開が複雑すぎる

既存のハイパーバイザーについて

一般的な解決策

- VirtualBox、KVMなど：type 2（ホスト）のため不適切
- Xen: 導入・展開が複雑すぎる

最低レベルの解決策

- bluepill、vitriol、virtdbg、hyperdbg など
- 「体内」仮想化、言いかえれば「立ち入り過ぎ」
- OSに依存

既存のハイパーバイザーについて

一般的な解決策

- VirtualBox、KVMなど：type 2（ホスト）のため不適切
- Xen: 導入・展開が複雑すぎる

最低レベルの解決策

- bluepill、vitriol、virtdbg、hyperdbg など
- 「体内」仮想化、言いかえれば「立ち入り過ぎ」
- OSに依存

最初からやり直してみる！

スタンドアロン型で「最低レベル」のハイパーバイザー

臨まれる仕様

- 簡単、軽い、早い、そして、安定
- 本来のパフォーマンスへの影響が小さい
- Intel-VT (vmx) とAMD- V (svm) の両方に対応
- 既存環境（BIOS）のメリットを享受できる
- 複雑なソフトウェアにならないようにシンプルな設計・実装を維持する
- ユーザーランド（userland）レベルのリモートコントロールで複雑な操作が可能

[1]実際には、Unrestricted Guest 機能にも対応

スタンドアロン型で「最低レベル」のハイパーバイザー

臨まれる仕様

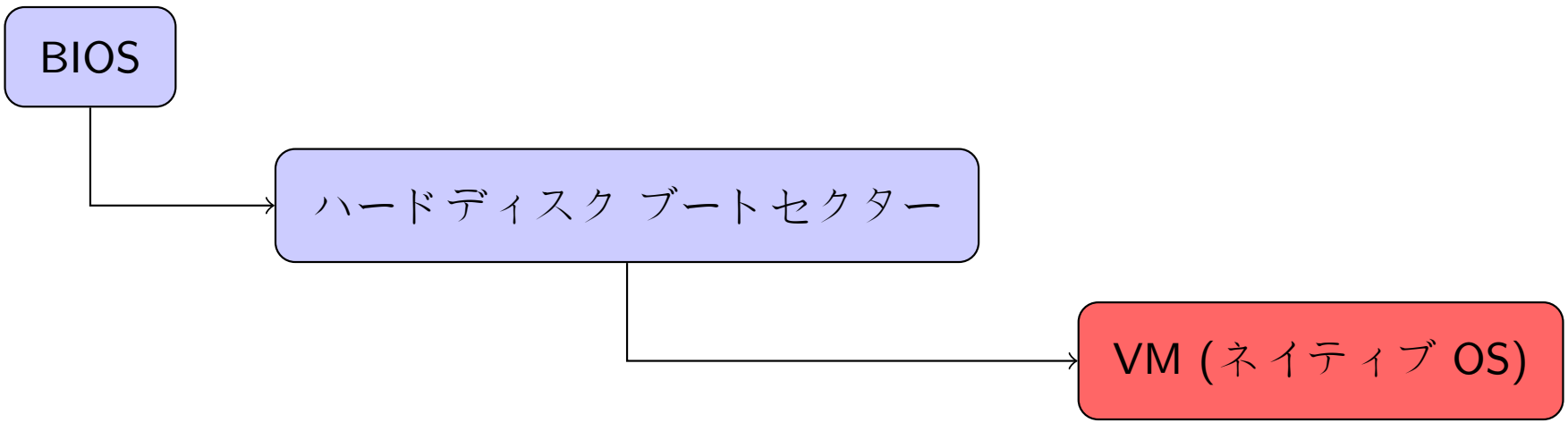
- 簡単、軽い、早い、そして、安定
- 本来のパフォーマンスへの影響が小さい
- Intel-VT (vmx) とAMD- V (svm) の両方に対応
- 既存環境（BIOS）のメリットを享受できる
- 複雑なソフトウェアにならないようにシンプルな設計・実装を維持する
- ユーザーランド（userland）レベルのリモートコントロールで複雑な操作が可能

最新のCPUをターゲットに

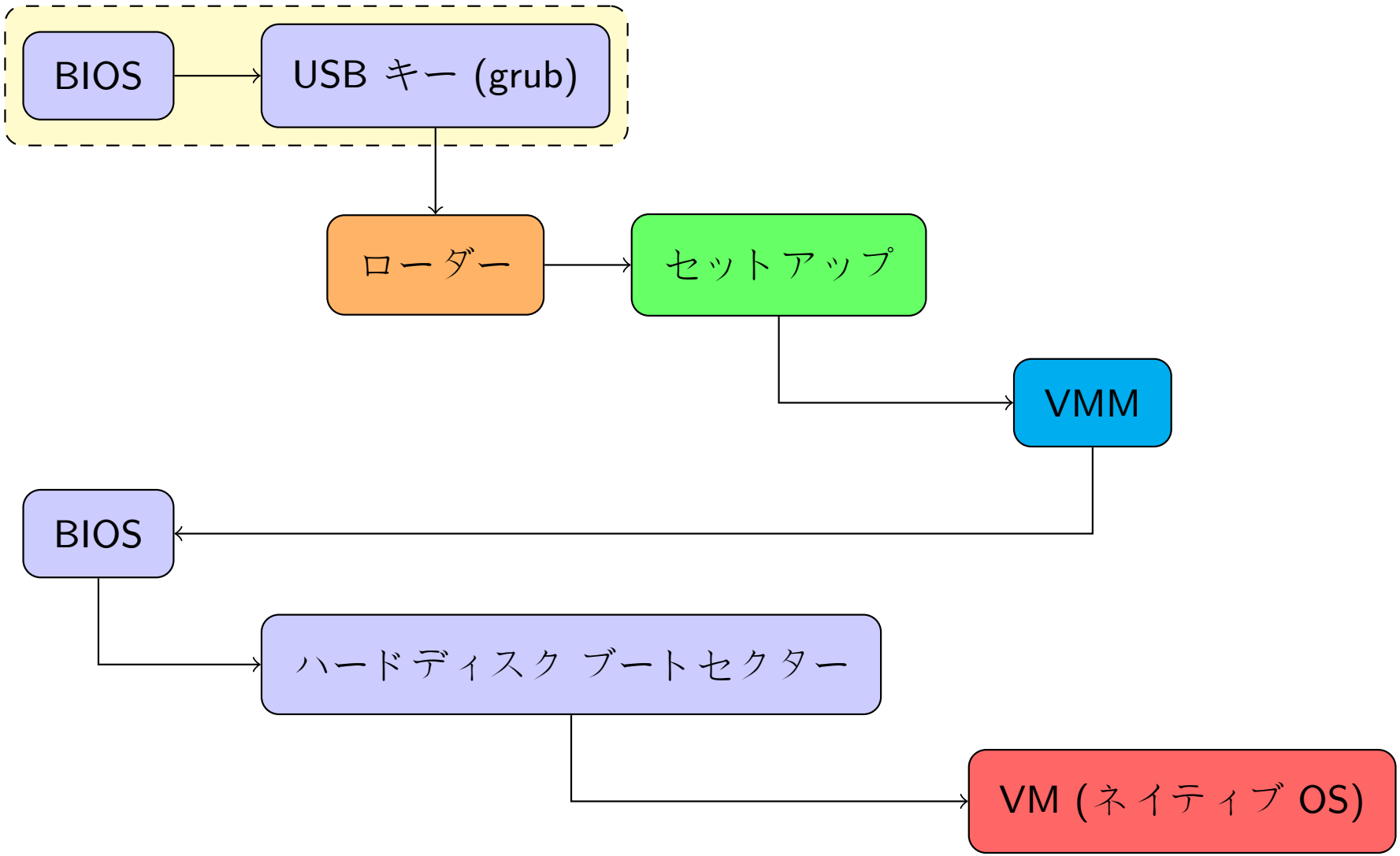
- 最近のハードウェア仮想化拡張機能を活用
- 特にIntel EPT[1] と AMD RVI
  - 簡単なコード
  - より早いVMM
  - 攻撃者に狙われる部分を少なく

[1]実際には、Unrestricted Guest 機能にも対応

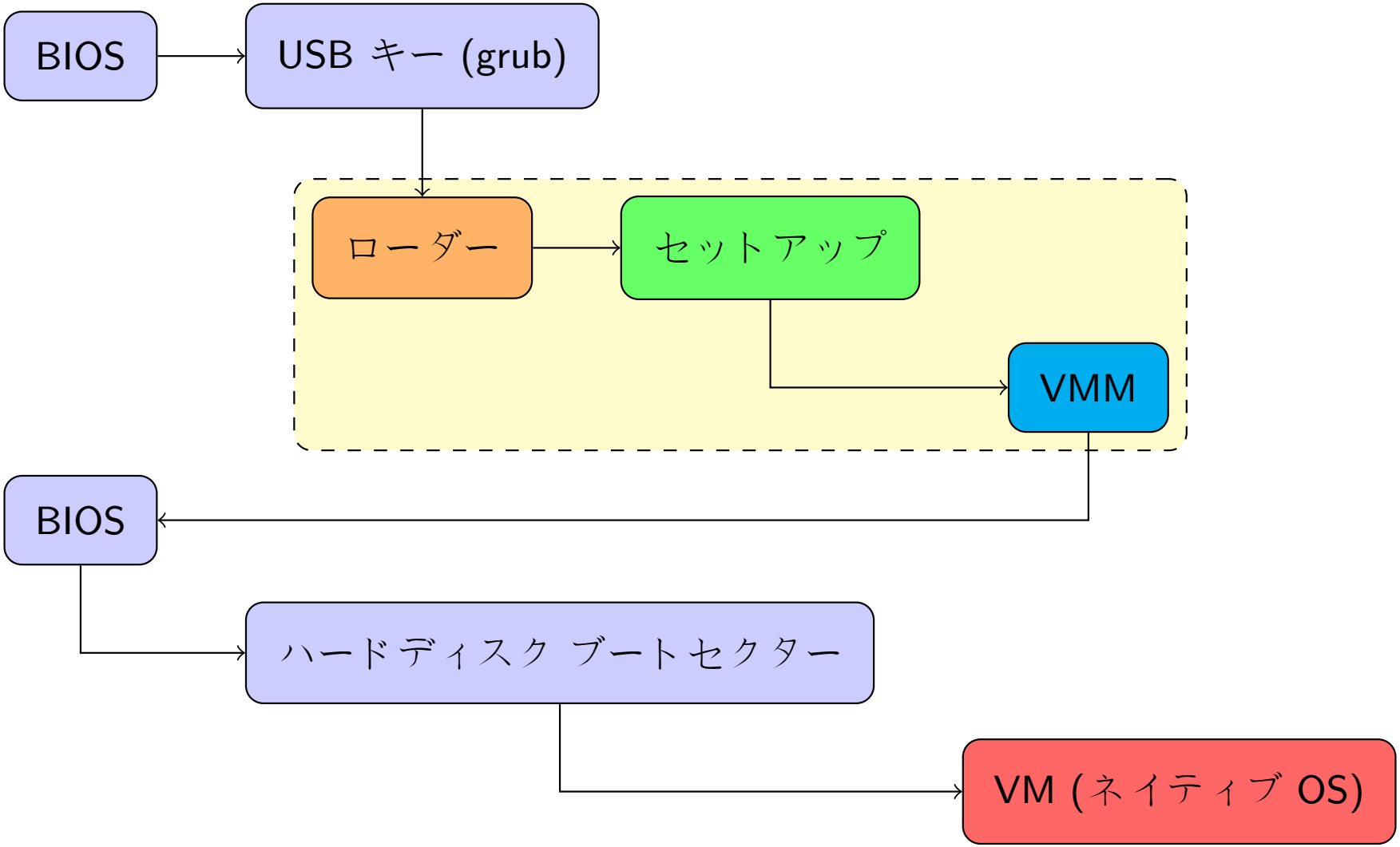
古典的なブート・シーケンス



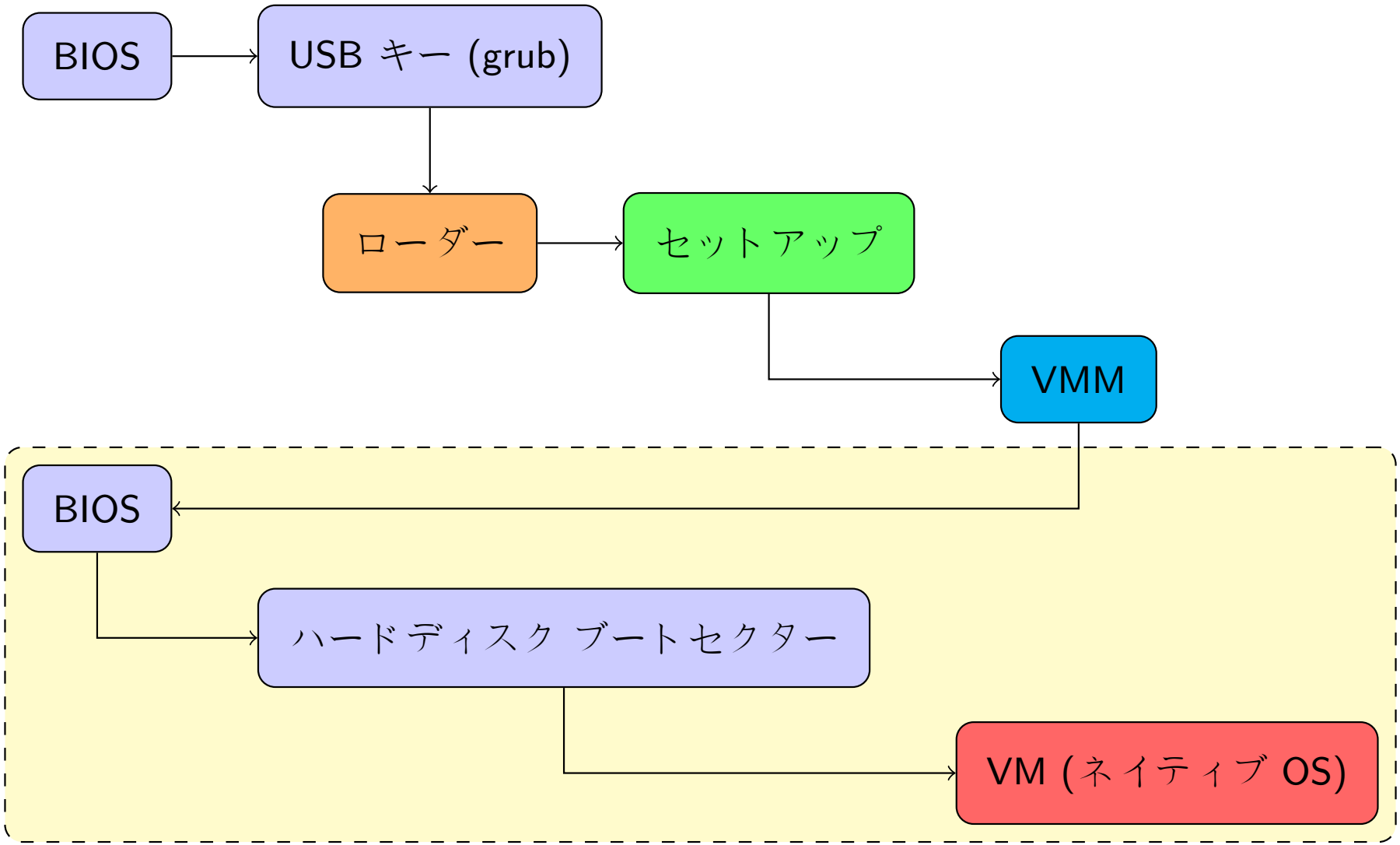
Ramooflax のブート・シーケンス



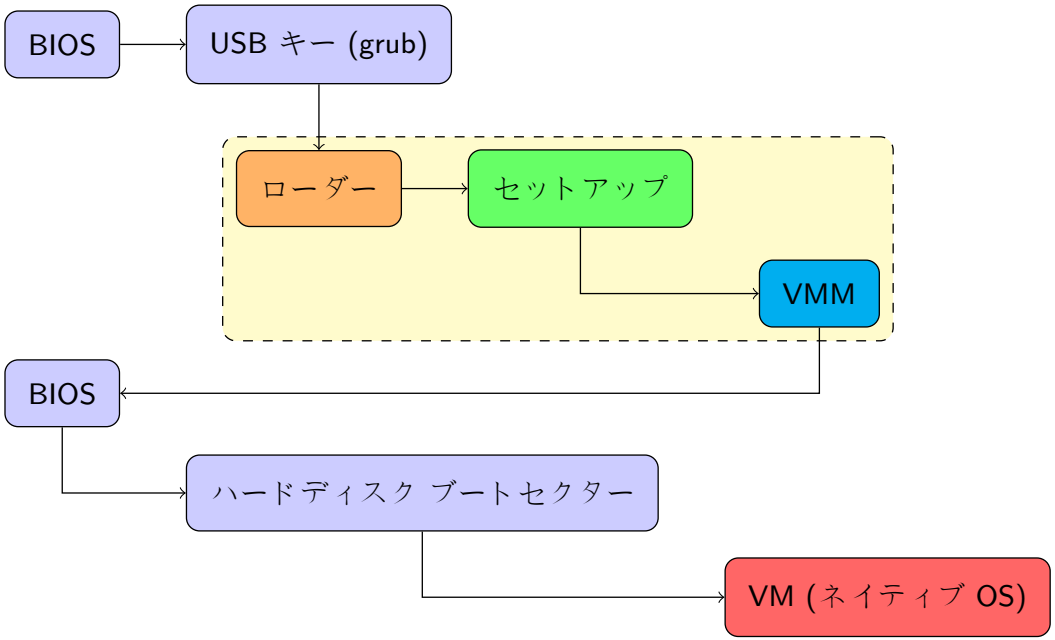
Ramooflax のブート・シーケンス



Ramooflax のブート・シーケンス



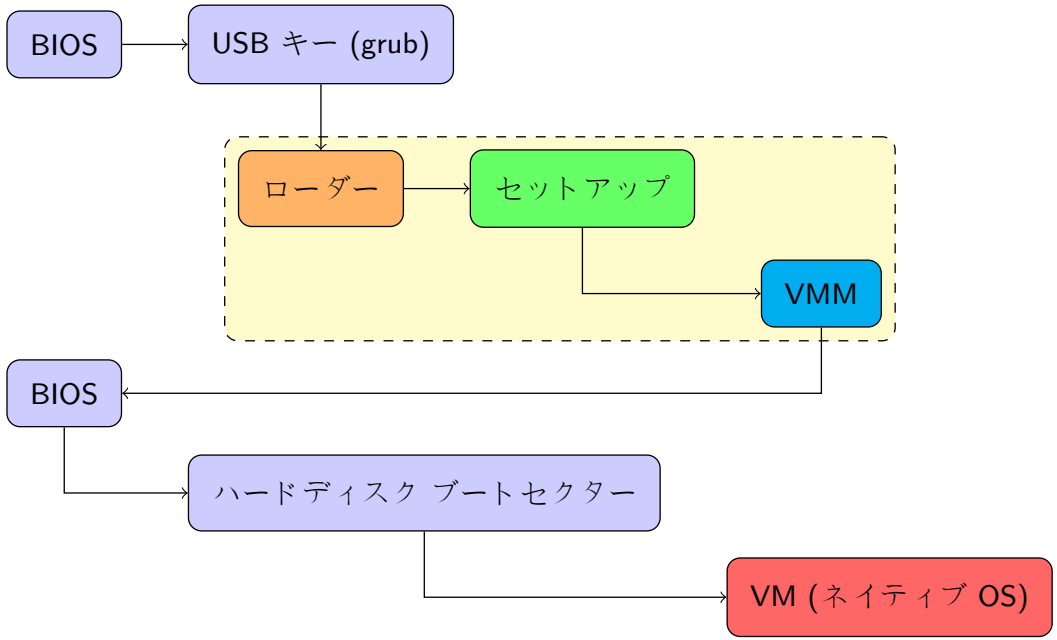
Ramooflax の各構成要素



ローダー

- 32ビットの保護モードでブート（multibootの標準仕様）
- longmode（64ビット）に切り替え後にセットアップをロード

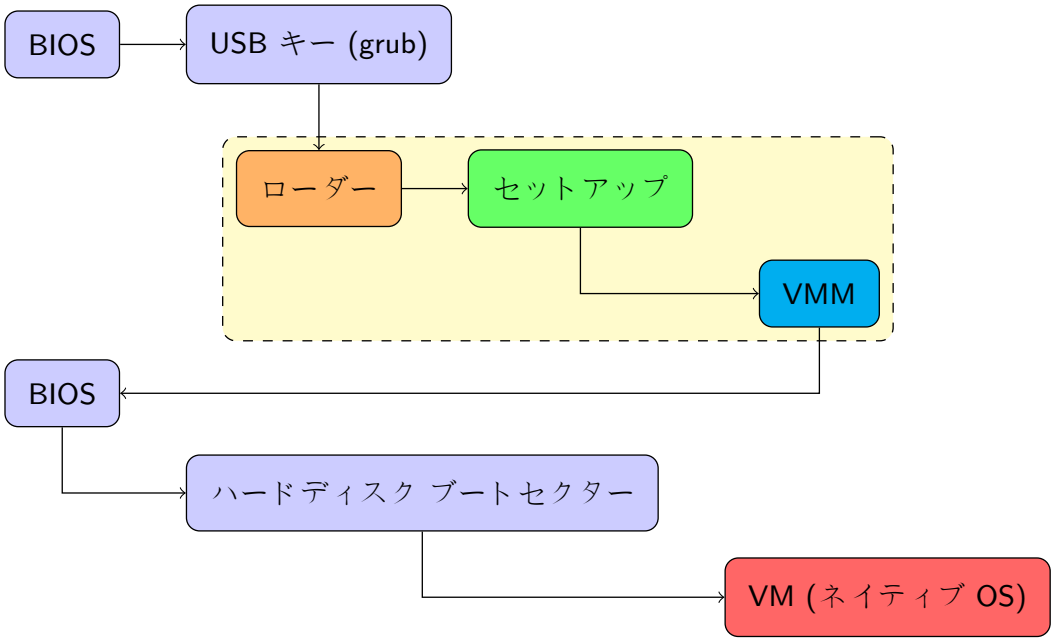
Ramooflax の各構成要素



セットアップ

- 仮想構造、ドライバ、メモリを初期化
- RAM容量を取得してVMMに必要なスペースを計算
- VMMを「size(RAM) - size(vmm)」に移動
- RAM容量を減少させる（環境に合わせたVM SMAPを作成）
- 「int 0x19」をコンベンショナル・メモリーにインストール
- VMMを起動

Ramooflax の各構成要素



VMMの動作状態

- PIEバイナリ（RAM容量は可変）
- リアルモードで単一のVMを「int 0x19」で起動
- BIOS（仮想化されている）にネイティブOSの起動を指示

Intel-VT (vmx) と AMD-V (svm) との共通点

注目の点

- ハイパーバイザーの開発を簡易化
- 命令セットの縮小（∼ 10)
- vm-entry/vm-exit の枠組み
  - vm-entryがVMをロードし、VMMを保存
  - vm-exitがVMMをロードし、VMを保存

データ構造の設定に依存

- AMD VMCB、 Intel VMCS（vmread, vmwrite が非同期）
- システムレジスタの設定 (cr, dr, gdtr, idtr, ...)
- イベント・インジェクション (interrupts, exceptions)
- インターセプション・ビットマップの設定
  - イベント
  - 機微な命令
  - I/O、MSRなどのアクセス

多くの制限事項

- インテルとAMDの互換性が必ずしも保証されない
- CPUモデル間での機能が異なる
- CPUスキルの学習は難しい（買って来でもしない限りは！

  `http://cpuid.intel.com?`

多くの制限事項

- インテルとAMDの互換性が必ずしも保証されない
- CPUモデル間での機能が異なる
- CPUスキルの学習は難しい（買って来でもしない限りは！

  `http://cpuid.intel.com?`

- vm-exit 処理後の情報が不足
- エミュレーション・ディスアセンブル エンジンを盛り込む必要が有る
- ハードウェア・ベースのインタラプト・インターセプションは on/off のみ。数値レベルの設定ができない。
- インテルはソフトウェア・ベースのインタラプト・インターセプションを供給していない
- AMDはハードウェア・ベースのインタラプトのリリースを保留
- SMI関連の問題（CPUのバグ、BIOSのバグ、SMMの仮想化が必要、等）

インテル環境下でリアルモードのマネジメントは非常に困難
実在のBIOS仮想化も非常に困難！

BIOS の仮想化

BIOSとリアル・モード

- 16ビット デフォルトCPUモード
- 20ビット（1MB）のメモリアドレス空間（プロテクション無し）
- ほとんどBIOSによって使用

旧モデルから受け継がれてきたリアルモード仮想化

- ハードウェアが支援する仮想化は80386から存在：v8086モード
- リアルモードのエミュレーション（interrupts, far call, . . .）
- redirect/intercept I/O が妨害

BIOS の仮想化

BIOSとリアル・モード

- 16ビット デフォルトCPUモード
- 20ビット（1MB）のメモリアドレス空間（プロテクション無し）
- ほとんどBIOSによって使用

旧モデルから受け継がれてきたリアルモード仮想化

- ハードウェアが支援する仮想化は80386から存在：v8086モード
- リアルモードのエミュレーション（interrupts, far call, . . .）
- redirect/intercept I/O が妨害

vmx/svmにおけるリアルモード仮想化

- AMDは新しくページングをサポートしたリアル・モードを提供 (`CR0.PE=0 && CR0.PG=1`)

BIOS の仮想化

BIOSとリアル・モード

- 16ビット デフォルトCPUモード
- 20ビット（1MB）のメモリアドレス空間（プロテクション無し）
- ほとんどBIOSによって使用

旧モデルから受け継がれてきたリアルモード仮想化

- ハードウェアが支援する仮想化は80386から存在：v8086モード
- リアルモードのエミュレーション（interrupts, far call, . . .）
- redirect/intercept I/O が妨害

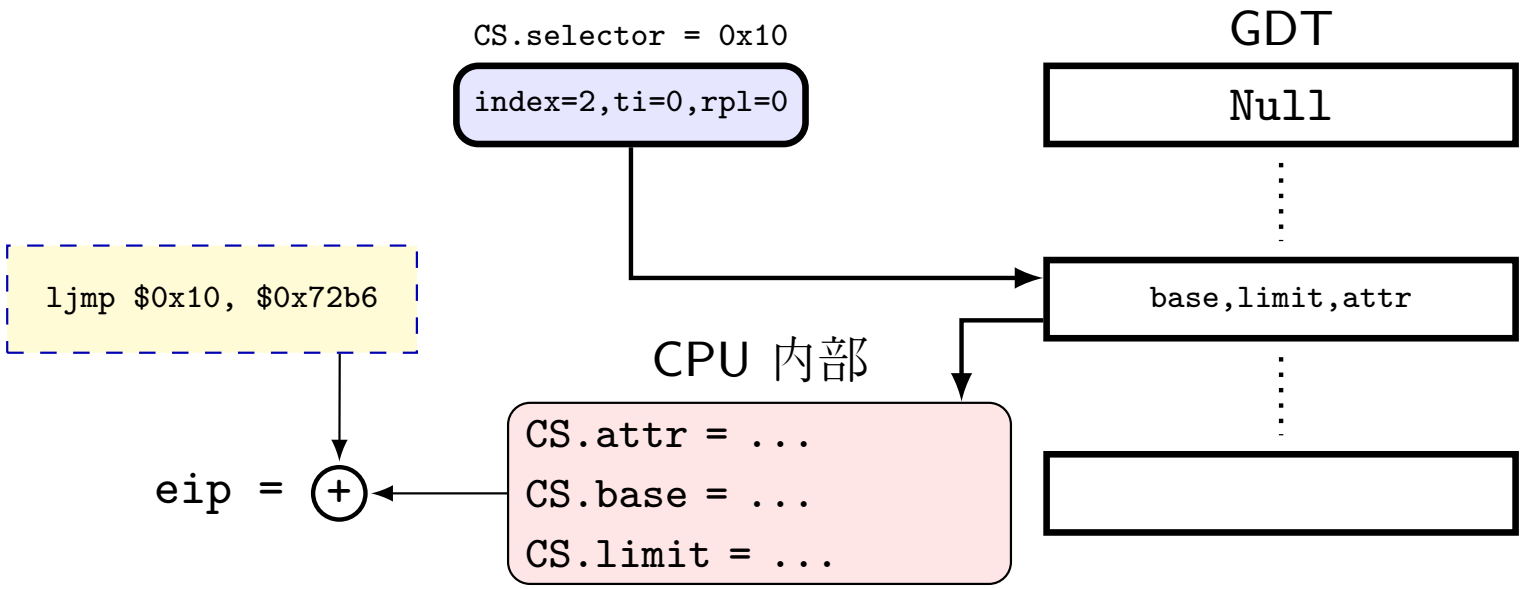
vmx/svmにおけるリアルモード仮想化

- AMDは新しくページングをサポートしたリアル・モードを提供 (`CR0.PE=0 && CR0.PG=1`)
- インテルは「CR0.PG=0」すなわち「CR0.PE=0」を許可しない
  - V8086モードの使用を推奨
  - V8086モードでのvm-entryには厳しい制限が有る
  - 特にセグメンテーションに対しては制限が厳しい

セグメンテーションについての注意ポイント

セグメント・レジスタ

- モニター可能な部分（selector）
- CPUによって管理された見えない部分 (base, limit, attributs)
- リアル・モード: $base = selector * 16$, $limit = 64K$
- プロテクテッド モード: segment descriptors



BIOSの仮想化

非リアル・モード（フラット・リアル、ビッグ・リアル）

- リアルモードで1MB以上のメモリにアクセス
- リアルモードへの移行を継続的に阻止: $base = 0$、$limit = 4GB$
- BIOSがメモリ空間にマップされたデバイスにアクセスするために使用

```
seg000:F7284          mov     bx, 20h
seg000:F7287          cli
seg000:F7288          mov     ax, cs
seg000:F728A          cmp     ax, 0F000h
seg000:F728D          jnz     short near ptr unk_7297
seg000:F728F          lgdt    fword ptr cs:byte_8163     (1)
seg000:F7295          jmp     short near ptr unk_729D
seg000:F7297          lgdt    fword ptr cs:byte_8169
seg000:F729D          mov     eax, cr0
seg000:F72A0          or      al, 1
seg000:F72A2          mov     cr0, eax                   (2)
seg000:F72A5          mov     ax, cs
seg000:F72A7          cmp     ax, 0F000h
seg000:F72AA          jnz     short near ptr unk_72B1
seg000:F72AC          jmp     far ptr 10h:72B6h          (3)
seg000:F72B1          jmp     far ptr 28h:72B6h
seg000:F72B6          mov     ds, bx                     (4)
seg000:F72B8          mov     es, bx
seg000:F72BA          mov     eax, cr0
seg000:F72BD          and     al, 0FEh
seg000:F72BF          mov     cr0, eax                   (5)
seg000:F72C2          mov     ax, cs
seg000:F72C4          cmp     ax, 10h                    (6)
seg000:F72C7          jnz     short near ptr unk_72CE
seg000:F72C9          jmp     far ptr 0F000h:72D3h
seg000:F72CE          jmp     far ptr 0E000h:72D3h
```

BIOSの仮想化

インテル上での失敗

- V8086モードでは、vm-entryが「$base = selector * 16$」をチェックする[1]
- V8086モードで非リアル・モードを仮想化できない

[1] Intel Volume 3B Section 23.3.1.2

BIOSの仮想化

インテル上での失敗

- V8086モードでは、vm-entryが「$base = selector * 16$」をチェックする[1]
- V8086モードで非リアル・モードを仮想化できない

ハードウェア仮想化の基本機能

- プロテクテッド・モードでのリアル・モードのエミュレーション
- セグメント・レジスタのアクセスを遮断: *far call/jump*, *mov/pop seg*, *iret*
- 二重の失敗: インテルはセグメント・レジスタの遮断機能を供給していない
- 解決策：GDTとIDTの値を0 に制限して#GPを遮断

[1] Intel Volume 3B Section 23.3.1.2

BIOSの仮想化

インテル上での失敗

- V8086モードでは、vm-entryが「$base = selector * 16$」をチェックする[1]
- V8086モードで非リアル・モードを仮想化できない

ハードウェア仮想化の基本機能

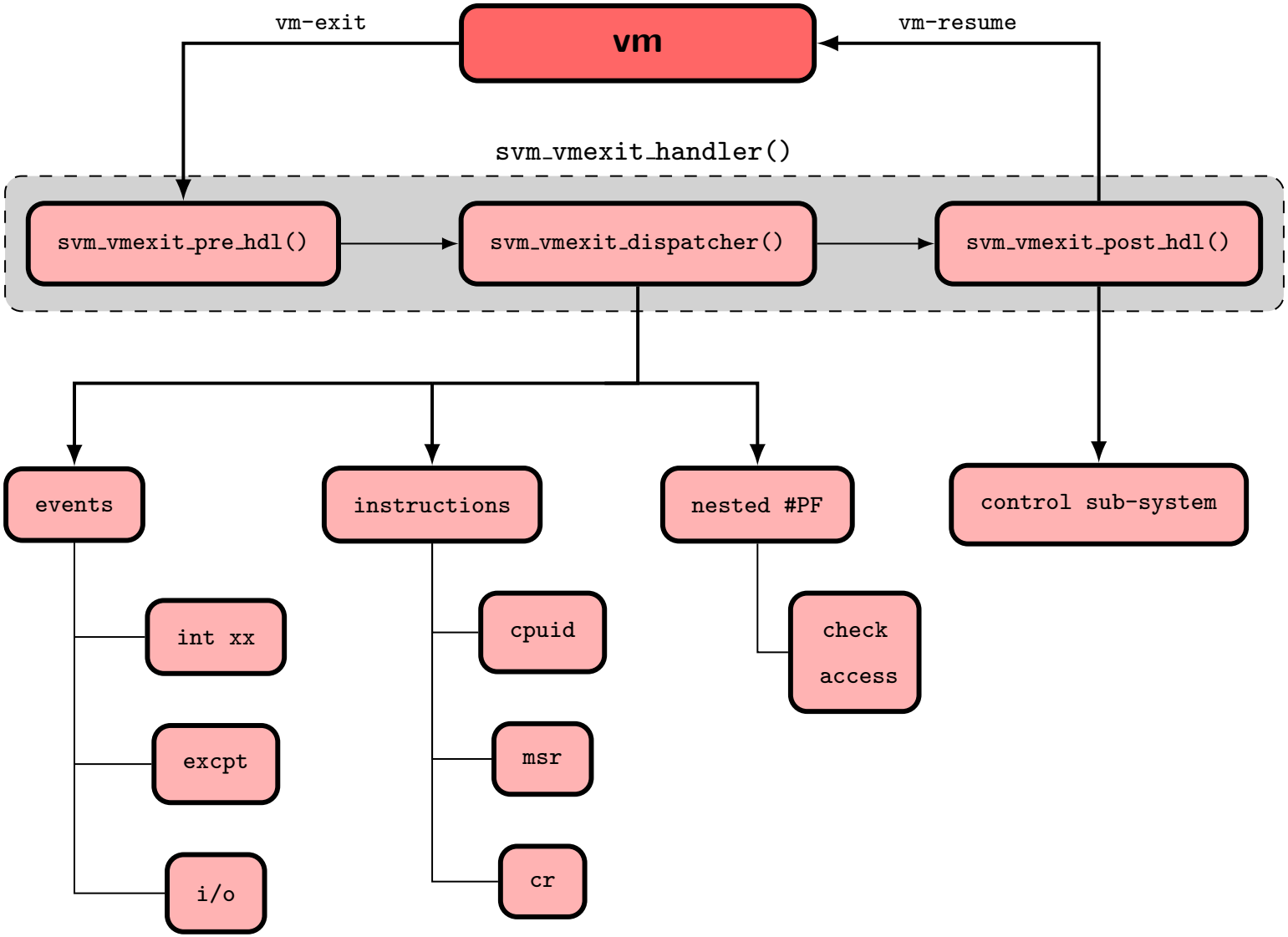
- プロテクテッド・モードでのリアル・モードのエミュレーション
- セグメント・レジスタのアクセスを遮断: *far call/jump*, *mov/pop seg*, *iret*
- 二重の失敗: インテルはセグメント・レジスタの遮断機能を供給していない
- 解決策：GDTとIDTの値を0 に制限して#GPを遮断

新型のCPUにおける付加機能

- 無制限のゲスト・モード (`CR0.PE=0 && CR0.PG=0` を許可)
- VMMメモリを保護するためにIntel EPTが必要

[1] Intel Volume 3B Section 23.3.1.2

実行フロー（AMDの場合）



システム・レジスタによるフィルタリング

コントロール・レジスタ

- cr0：モード移行、キャッシュの整合性維持、メモリのマッピング
- cr3：リモート制御（詳細は後述）
- リモートクライアント機能としても活用

MSRとCPUIDの読み取り

- 通常もしくはバックエンドでのVMCS/VMCB読み取り
- 特有の機能を隠すための後処理

MSRの書き込み

- VMCS/VMCB をバックエンドで事項している場合にwrmsrをエミュレート
- その他の通常実行

イベントのフィルタリング

例外処理

- 主にサブシステム制御のための#DBと#BPに対するきめ細かい遮断処理
- インテル環境において特有のソフトウェアベースの割り込みを遮断するための#GPフィルタリング

ソフトウェアベースの割り込み

- only in real mode
- SMAPのアクセス(int 0x15)をフィルタリング

ハードウェアベースの割り込み

- 遮断しない
- しかし、遮断は可能

エミュレーション

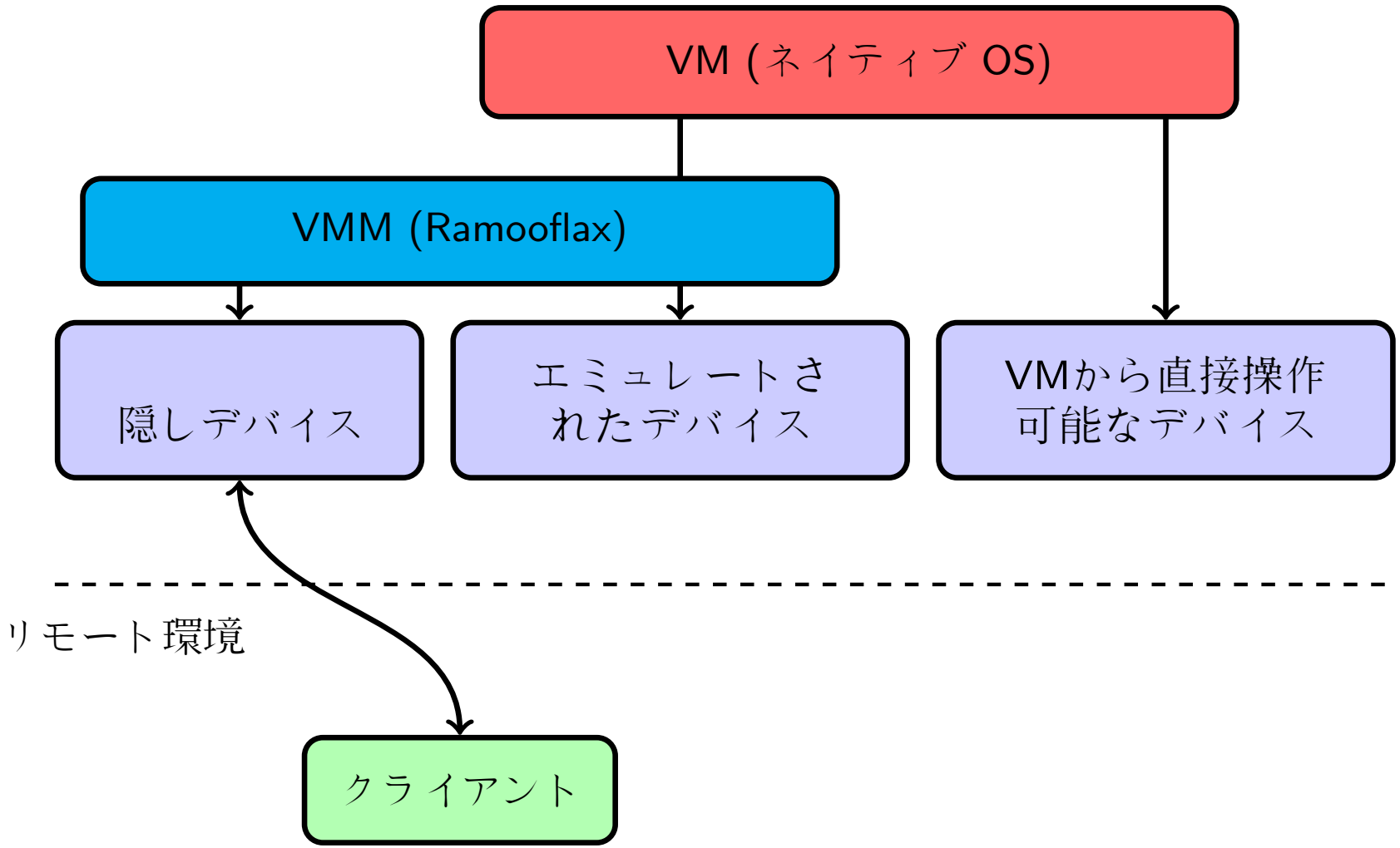
命令群

- ディスアセンブルおよびエミュレーションにはvm-exitを適切に扱う必要が有る
- Ramooflax はudis86を組み込み...やり過ぎた？
- エミュレートされた命令群はシンプルなもの
- 実行Contextを管理

デバイス

- UART、PIC、KBDおよびPS2 システムコントローラーの部分的エミュレーション・遮断
- リブートビットの制御が主な目的

VMM、VM、および、クライアントの動作フロー



リモートからの通信

UART

- 遅くて信頼性が低い
- デバッグの目的のみに使用

EHCIデバッグポート

- USB 2.0の仕様ではUSBの物理ポートがデバッグポートとして使用可能
- 多くのEHCIホスト・コントローラで使用されている
- 信頼性が高く、標準化されていて、早い
- UART並みに使用が簡単

Ramooflax 側の実装

- デバッグ・ポートのドライバ
- VM制御化でも使用可能なEHCIホスト・コントローラ

リモート通信

クライアント側のEHCIデバッグ・ポート

- USBの使用：ホストコントローラー間で直接のデータやり取りを行わない
- デバッグ・デバイスが必要
  - 専用のデバイスを購入（例：Net20DC）
  - 「USB On-The-Go」コントローラーのメリットを活用（スマートフォンなど）

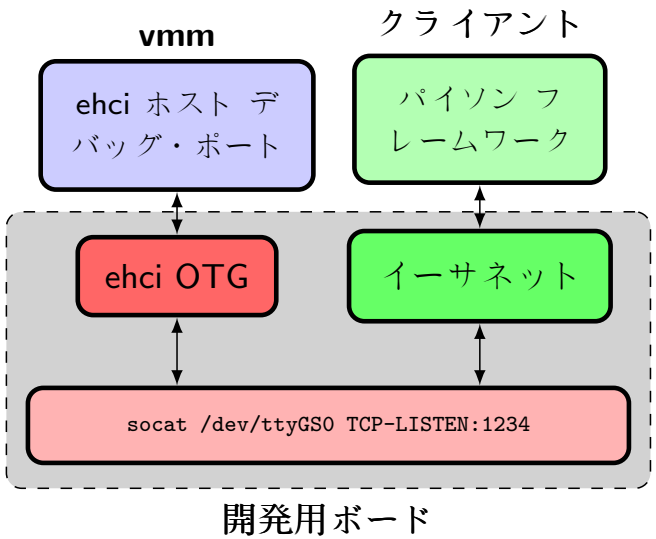
リモート通信

クライアント側のEHCIデバッグ・ポート

- USBの使用：ホストコントローラー間で直接のデータやり取りを行わない
- デバッグ・デバイスが必要
  - 専用のデバイスを購入（例：Net20DC）
  - 「USB On-The-Go」コントローラーのメリットを活用（スマートフォンなど）

Linux下でのデバッグ・デバイスのエミュレーション

- ガジェットAPIがUSBデバイスのエミュレーションを許可（大容量記憶装置等）
- デバッグ・デバイスのガジェット実装はシリアルインターフェースを暴露



クライアント制御

コントロール権を取得

- VMMはwm-exitを待っている
- クライアントの反応とVMのパフォーマンスのトレードオフを検出
- VMMがクライアントからの要求に基づいてVMのコントロール権を確実に取得できるようにする
- 最近、インテルはvmx_preemption_timer をリリース、しかし、AMDには無し。

ハードウェアからの割り込みを通じて？

- デバッグポートに挙げられるirqは無い
- 複雑で表面には出てこない

コンテキスト・スイッチ

- 最近のOSにおけるスケジューリング手法
- 割り込みはcr3に書き込み

クライアント制御

GDBスタブの実装

- 汎用レジスタへの読み書き
- メモリへの読み書き
- ソフトウェアおよびハードウェアベースのブレークポイントの設定と削除
- シングルステップ実行

プロトコルの制限

- ユーザスペースで動くアプリケーションのデバッグ用に作成
- ring0情報（セグメンテーション、ページングなど）無し
- 仮想メモリ・物理メモリの識別をしない

クライアント制御

Ramooflax 特有の実装

- システムレジスタへのアクセス
  - cr0, cr2, cr3, cr4
  - dr0-dr3, dr6, dr7, dbgctl
  - cs, ss, ds, es, fs, gs ベースアドレス
  - gdtr, idtr, ldtr, tr

クライアント制御

Ramooflax 特有の実装

- システムレジスタへのアクセス
  - cr0, cr2, cr3, cr4
  - dr0-dr3, dr6, dr7, dbgctl
  - cs, ss, ds, es, fs, gs ベースアドレス
  - gdtr, idtr, ldtr, tr
- メモリアクセス
  - 仮想メモリと物理メモリを識別
  - アドレス・トランスレーション機構
  - 固定cr3機能（VMMに特定のcr3を保持した状態での動作を強制）

クライアント制御

Ramooflax 特有の実装

- システムレジスタへのアクセス
  - cr0, cr2, cr3, cr4
  - dr0-dr3, dr6, dr7, dbgctl
  - cs, ss, ds, es, fs, gs ベースアドレス
  - gdtr, idtr, ldtr, tr
- メモリアクセス
  - 仮想メモリと物理メモリを識別
  - アドレス・トランスレーション機構
  - 固定cr3機能（VMMに特定のcr3を保持した状態での動作を強制）
- 仮想化制御
  - コントロールレジスタの割り込み
  - 例外割り込み
  - 完全に . . . VMCS/VMCBへのフルコントロール

クライアント制御

シングルステップ実行

- TFおよび例外割り込みに基づく
- 単一のVM下で様々実行モード
  - グローバル（実装済み）
  - カーネル・スレッドのみ
  - ring 3 プロセスのみ（実装済み）
  - ring 0/3プロセスの見（システムコールを追跡）
- 仮想化されたOSのコンセプト関連の機能は無し（プロセス修了）
- ステルス・整合性維持(pushf,popf,intN,iret 割り込み)

クライアント制御

シングルステップ実行

- TFおよび例外割り込みに基づく
- 単一のVM下で様々実行モード
  - グローバル（実装済み）
  - カーネル・スレッドのみ
  - ring 3 プロセスのみ（実装済み）
  - ring 0/3プロセスの見（システムコールを追跡）
- 仮想化されたOSのコンセプト関連の機能は無し（プロセス修了）
- ステルス・整合性維持(pushf,popf,intN,iret 割り込み)

特別のケース: `sysenter/sysexit`

- AMDでもインテルでも（!!!）割り込み不能
- ring 0に入る時にはTFをマスクしない
- フォールト・ベースのメカニズム実装が必要（インテルのソフトウェアによる割り込みが行われるため）

ハイパーバイザーへのパイソン・インターフェース

フレームワーク構成品

- VM：限られた機能のみ
- CPU：レジスタ、例外処理のフィルタリング
- ブレークポイント：ソフトウェア ベース・ハードウェア ベース
- GDB：Ramooflax 拡張機能付きのGDBクライアント
- メモリ：メモリアクセスを生後y
- イベント：オリジナルのパイソン・ハンドラを実装するためのvm-exitフッキング機構

フレームワーク構成品：VM

- 実行、停止、再開、シングル・ステップ実行、アタッチ、デタッチ

```
vm = VM(CPUFamily.AMD, "192.168.254.254:1234")
```

- インタラクティブ・モード

```
vm.run(dict(globals(), **locals()))
```

- スクリプト・モード

```
vm.attach()  # remote connection
vm.stop()    # stop it

# xxxx (breakpoints, filters, ...)

vm.resume()  # resume and wait for next vm-exit
vm.detach()  # disconnect, vm resumed
```

フレームワーク構成品：CPU、メモリ、ブレークポイント

- ブレークポイントの命名

```
# data write breakpoint
vm.cpu.breakpoints.add_data_w(vm.cpu.sr.tr+4, 4, filter, "esp0")

>>> vm.cpu.breakpoints
esp0 0xc1331f14 Write (4)
kernel_f1 0xc0001234 eXecute (1)
```

- cr3トラッキング機能

```
# reading a virtual memory page
vm.cpu.set_active_cr3(my_cr3)
pg = vm.mem.vread(0x8048000, 4096)
```

フレームワーク構成品：イベント

- GDB条件付きのブレークポイント文法 ...
- 開発者にvm-exit 後の関数実行を許可
- アークテクチャやOS特有の気候を分離
- 例外をフィルタリング：cr3への書き込み、ブレークポイント など ...

```
def handle_excp(vm):
   if vm.cpu.gpr.eip == 0x1234:
     return True
   return False

vm.cpu.filter_exception(CPUException.general_protection, handle_excp)

while not vm.resume():
   continue

vm.interact()
```

結論

サポート内容

- AMDとインテルをサポート
- 下記のOSでテストが成功
  - Windows XP/7 Pro 32 ビット
  - Debian GNU/Linux 5.0 32/64 ビット
- より簡易なOS（DOSやOpenBSD）は実行できるはず

結論

サポート内容

- AMDとインテルをサポート
- 下記のOSでテストが成功
  - Windows XP/7 Pro 32 ビット
  - Debian GNU/Linux 5.0 32/64 ビット
- より簡易なOS（DOSやOpenBSD）は実行できるはず

制限事項

- 最近のCPU（Phenom II, Westmer/Sandy bridge）は対応せず
- SMPとマルチコアもサポートせず
  - セットアップが非常に複雑
    - すべてのコアを初期化して仮想化
    - コアの初期化の割り込みがVMによって行われる
  - 回避策
    - BIOSの設定
    - カーネルのパラメータ設定 `/numproc`, `maxcpus`
- 入れ子状態の仮想化には対応せず

Thank you !

https://github.com/sduverger/ramooflax