# セグメントアーキテクチャ計算機向け仮想マシンモニタの仮想化システム

Virtulization system including a virtual machine monitor for a computer with a segmented architecture

**作成 清水正明** 2004/10/26

**米国特許**
**特許番号**：US6,397,242 B1
**特許出願日**：1998**年**10**月**26**日**
**特許付与日**：2002**年**5**月**28**日**
**発明者**：Scott W. Devine**他**
**権利人**：VMware, Inc.

# 目次

- 既存の技術と問題点
- Intel x86アーキテクチャと仮想化における問題
- 発明の概要
- 機能の説明
  - 実行モードの選択
  - 直接実行サブシステム
  - バイナリ変換サブシステム
  - セグメントの仮想化
  - メモリトレース
- 本発明によるシステムの例

# 既存の技術

- ニーズ：別のＯＳ用に作成されたアプリケーションを実行したい

  - 仮想マシンモニタ
  - マシンシミュレータ/エミュレータ
  - アプリケーションエミュレータ
  - オペレーティングシステムエミュレータ
  - 同居型オペレーティングシステム
  - 過去互換性のための仮想マシンモニタ
  - ブートマネージャ

# 既存の技術

## 仮想マシンモニタ

- 1960年代末から1970年代にかけて熱心に研究された
  IBMもVM/370で仮想マシンを採用
- ハードウェアとOSの間の薄皮インタフェース
- “直接実行”向きのプロセッサアーキテクチャを開発

↑

- ハードウェアが高価だった
- OSがシングルユーザだったので複数人で使用するために
  VMMで多重化した

↓

ハードウェアのコストダウン＆ＯＳの機能向上で魅力低減

しかし近頃、フォールトトレランス、SMP上でコモディティＯＳを実行するために再び脚光

# 既存の技術

## 仮想マシンモニタ

- 異なるプロセッサアーキテクチャ間で互換性を提供
  ↓
  バイナリ変換

- IBM DAISY
  VLIWアーキテクチャ上でPowerPCとx86をエミュレート
  ホストOSなしで動作

# 既存の技術

## マシンシミュレータ/エミュレータ

- ホストOS上で動作する
- 計算機システムのすべての資源をエミュレート
- エミュレーション技術はバイナリ変換

- ホストOS上で動作する利点
  デバイスドライバ等のＯＳの機能が使える
- ホストＯＳ上で動作する欠点
  特権命令やアドレス空間操作が制限されることがある

**例**：VirtualPC, RealPC

# 既存の技術

## アプリケーションエミュレータ

アプリケーション層のプログラムに対して異なるプロセッサアーキテクチャ間の互換性を提供する。

バイナリ変換＋ターゲットＯＳのシステムコールへの変換

例:

68000向けバイナリをPowerPCベースのMacintoshで実行

x86 WindowsNTのアプリケーションをAlpha WindowsNTで実行

Macintoshアプリケーションを Linux上のExecuterで実行

Macintoshアプリケーションを Unix上のMAEで実行

# 既存の技術

## オペレーティングシステムエミュレータ

- あるオペレーティングシステムアプリケーションバイナリインタフェース(ABI)で書かれたアプリケーションを、他のABIのOSで実行できるようにシステムコールを変換する。

- 対象とするOSに強く依存

- アプリケーションエミュレータとの違い
  プロセッサアーキテクチャの変換はしない

例:
Windowsアプリケーションを Linuxで実行(WINE)
UnixアプリケーションをWindowsNT上で実行

# 既存の技術

## 同居型オペレーティングシステム

・ホストＯＳが提供していない確実性を追加してアプリケーション
　に提供する

例：
　WindowsNT + リアルタイムカーネル
　(NTプロセスとリアルタイムプロセスの同居)

制限
　・ハードウェアアブストラクションレイヤー (HAL) 等、ＯＳの
　　最下層の変更が必要 (割り込みのtrap等)
　　→対象ＯＳに強く依存 (独立性が低い)

# 既存の技術

## 過去互換型仮想マシンモニタ

・プロセッサの過去互換モードを使用して、古いＯＳを動作させる

例：Windows上+x86のv8086モードでMS-DOSを実行

既存のＯＳ上や、ＯＳに組み込まれているため、本物の仮想マシンモニタとは異なる

# 既存の技術

## ブートマネージャ

・ハードディスクのパーティションを切り替えることでＯＳを
切り替えることが可能。
↓
ただし、再起動が必要なので同時に異なるＯＳの複数の
アプリケーションを動作させることはできない

# 過去の技術の問題点

・仮想マシンはオーバーヘッドやホストＯＳの制限を受けない

↓

性能や一般性、同時実行性でアドバンテージがある。

しかし、厳密に仮想化可能の場合は“直接実行”ができるが、
x86のように“厳密には仮想化できない”プロセッサに対しては
バイナリ変換を用いる二者択一しかできない。

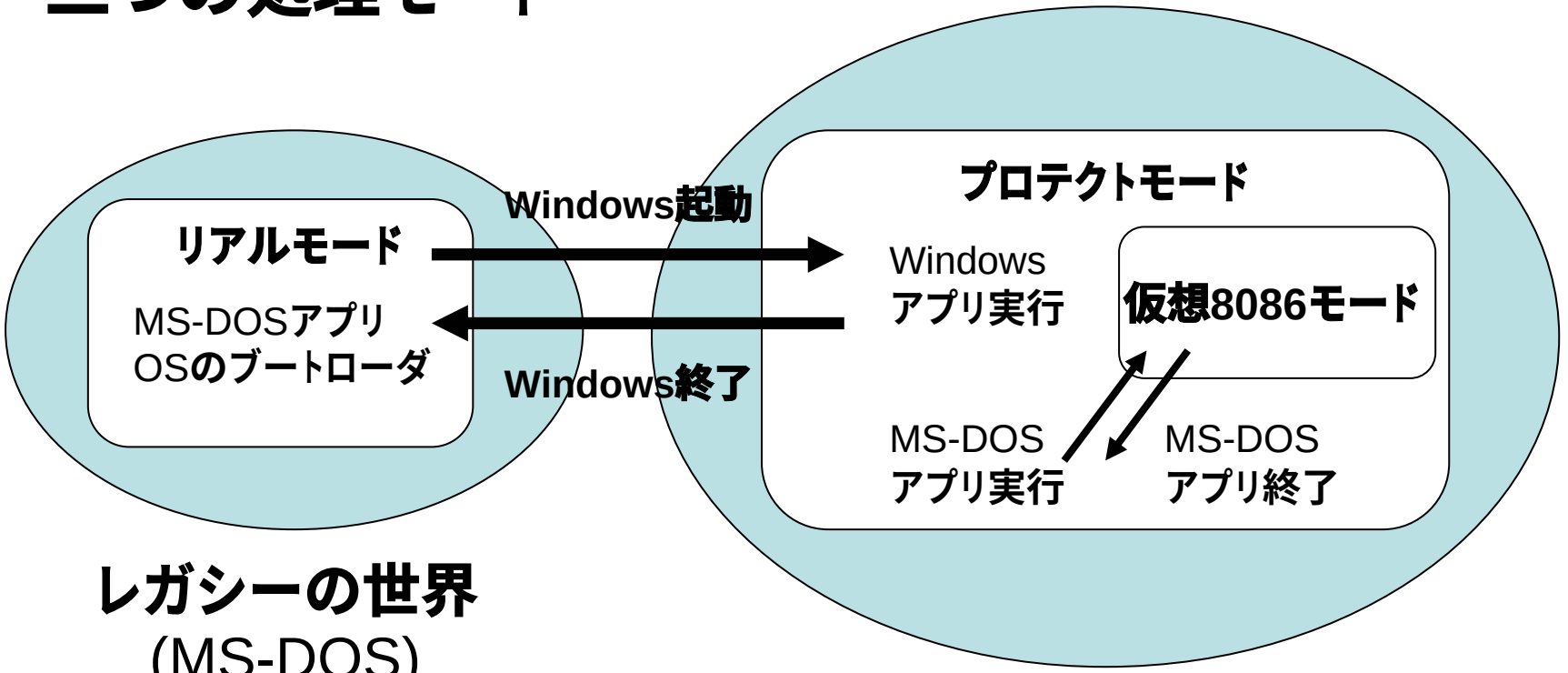
・ゆえに必要とされるのは、直接実行のスピードとバイナリ変換の
自由度の両方を持つVMMである。また効率よく二つのモードを
切り替える機能を持つ必要がある。

↓

本発明はこのようなシステムを提供する

# Intel x86アーキテクチャと仮想化における問題

## x86アーキテクチャの概要
## ・三つの処理モード




**レガシーの世界**
(MS-DOS)

**現代の世界**
(Windows, Linux)

# Intel x86アーキテクチャと仮想化における問題

## プロテクトモードにおける機能

- セグメンテーション
  仮想アドレスからリニアアドレスへの変換

- ページング
  リニアアドレスから物理アドレスへの変換

- 4つの特権レベル
  0：システムレベル、3：ユーザレベル

# Intel x86アーキテクチャと仮想化における問題

## 仮想化不可能命令の存在

OSを仮想マシン上で動作させるためには、通常は特権モードで動作しているOSをユーザモードで実行する必要がある。

↓

OSに気づかれずに騙したい

↓

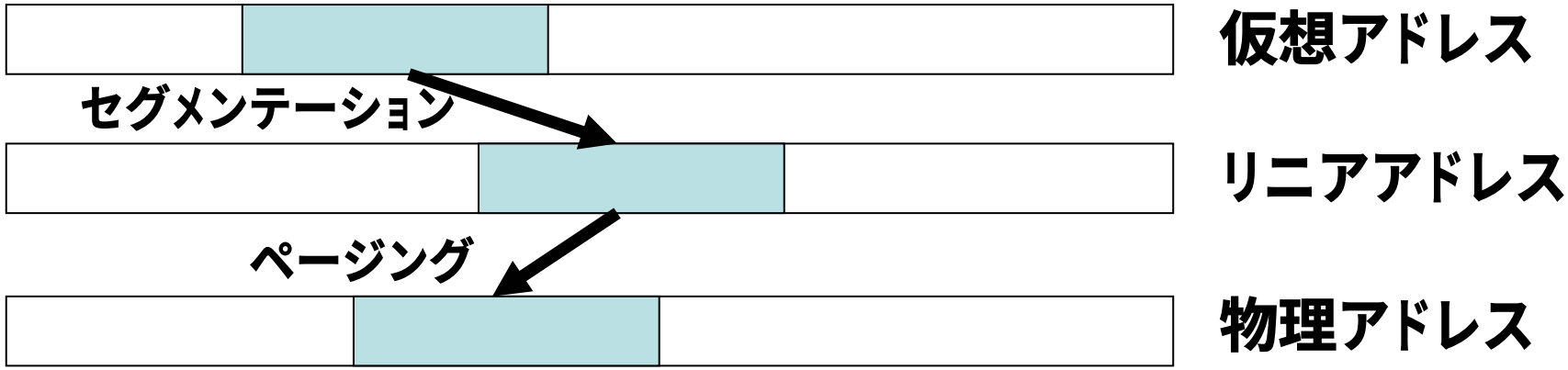
特権モード/ユーザモードを判定できる命令をTrapしてエミュレートしてしまえばよい（特権命令エミュレーション）

↓

しかしIntel x86にはユーザモードで特権レベルが確認できる命令があり、trap不可能(IRET, PUSHF)。

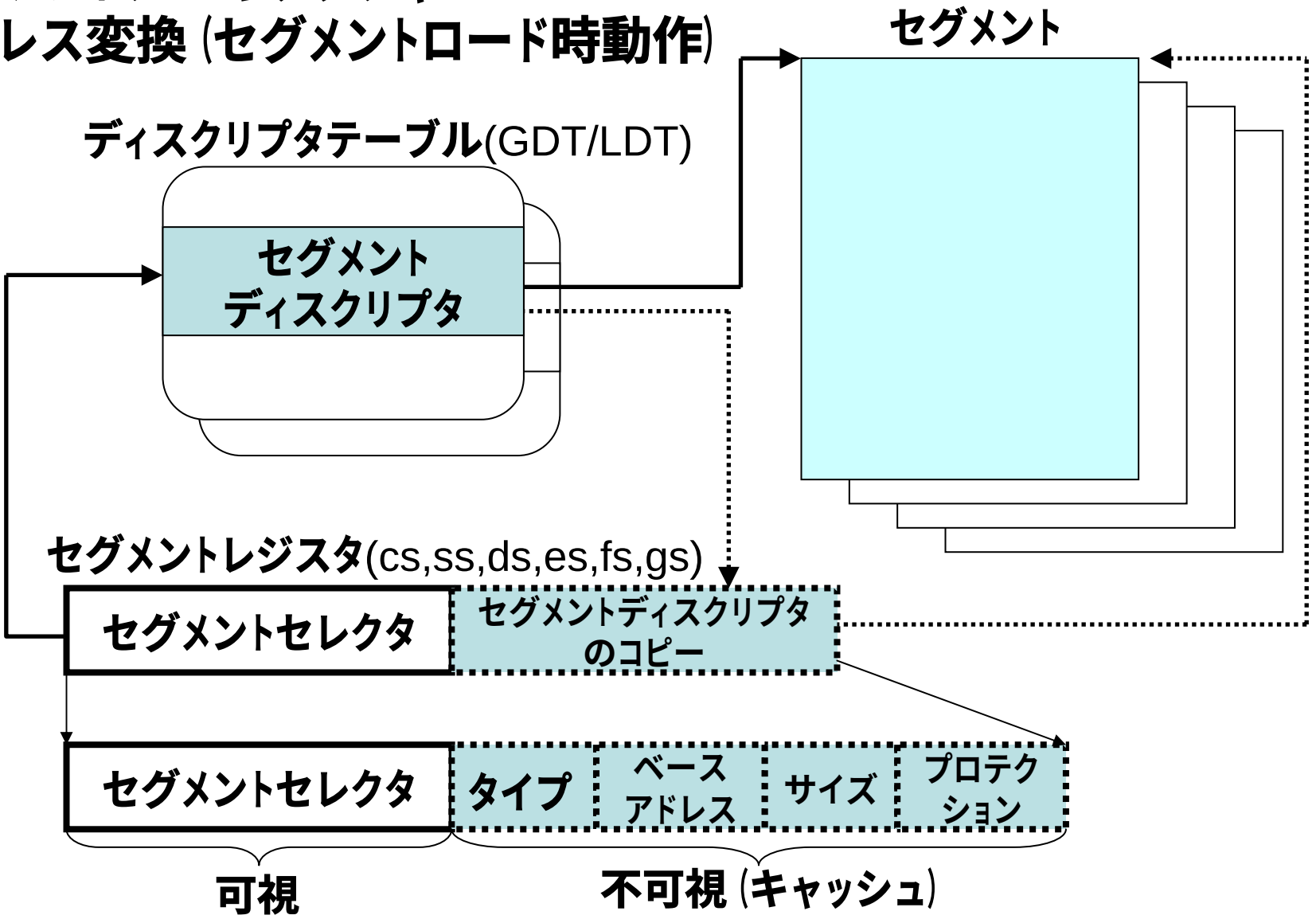
# Intel x86アーキテクチャと仮想化における問題

## セグメントアーキテクチャ
## アドレス変換



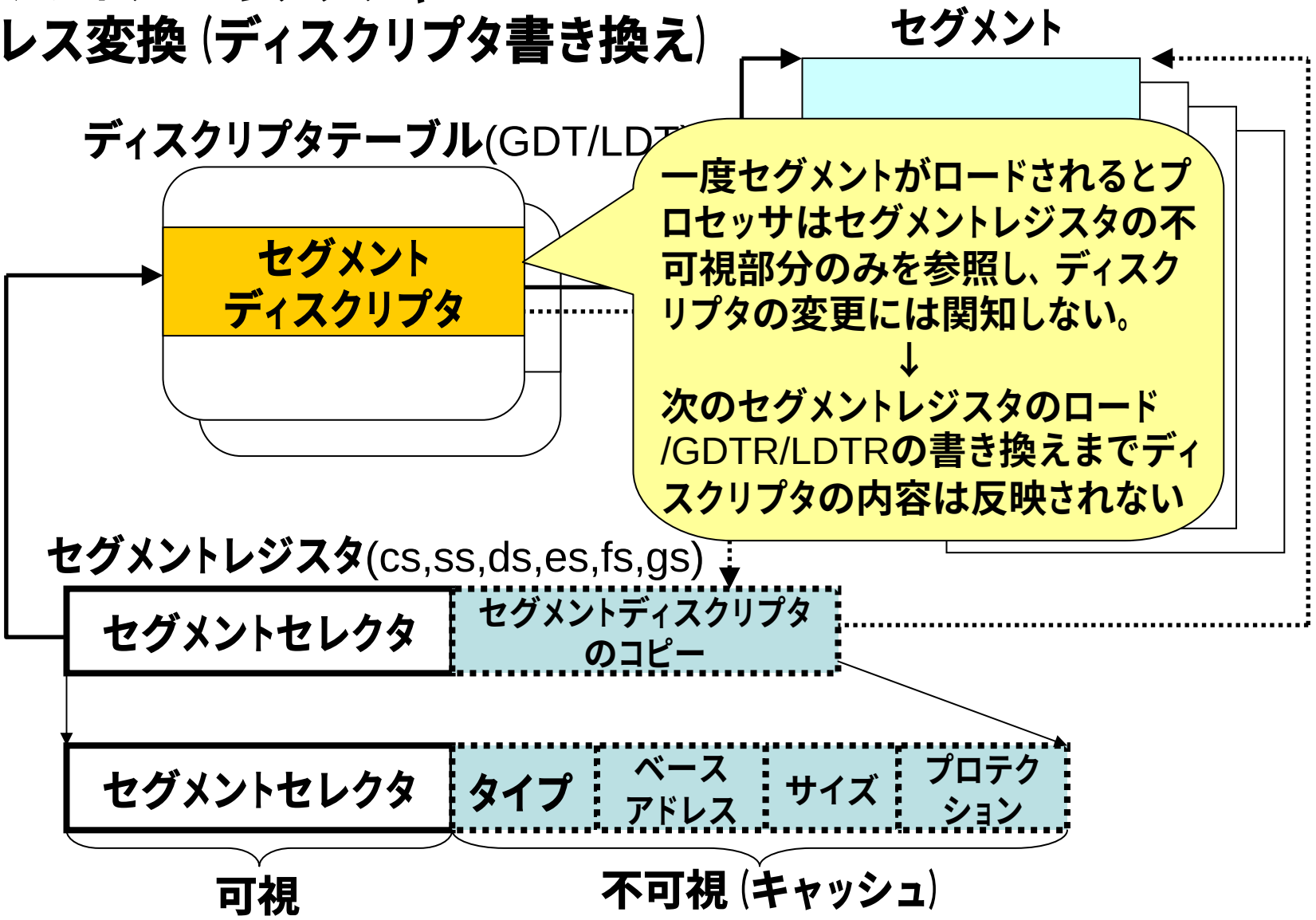
# Intel x86アーキテクチャと仮想化における問題

## セグメントアーキテクチャ
## アドレス変換 (セグメントロード時動作)



# Intel x86アーキテクチャと仮想化における問題

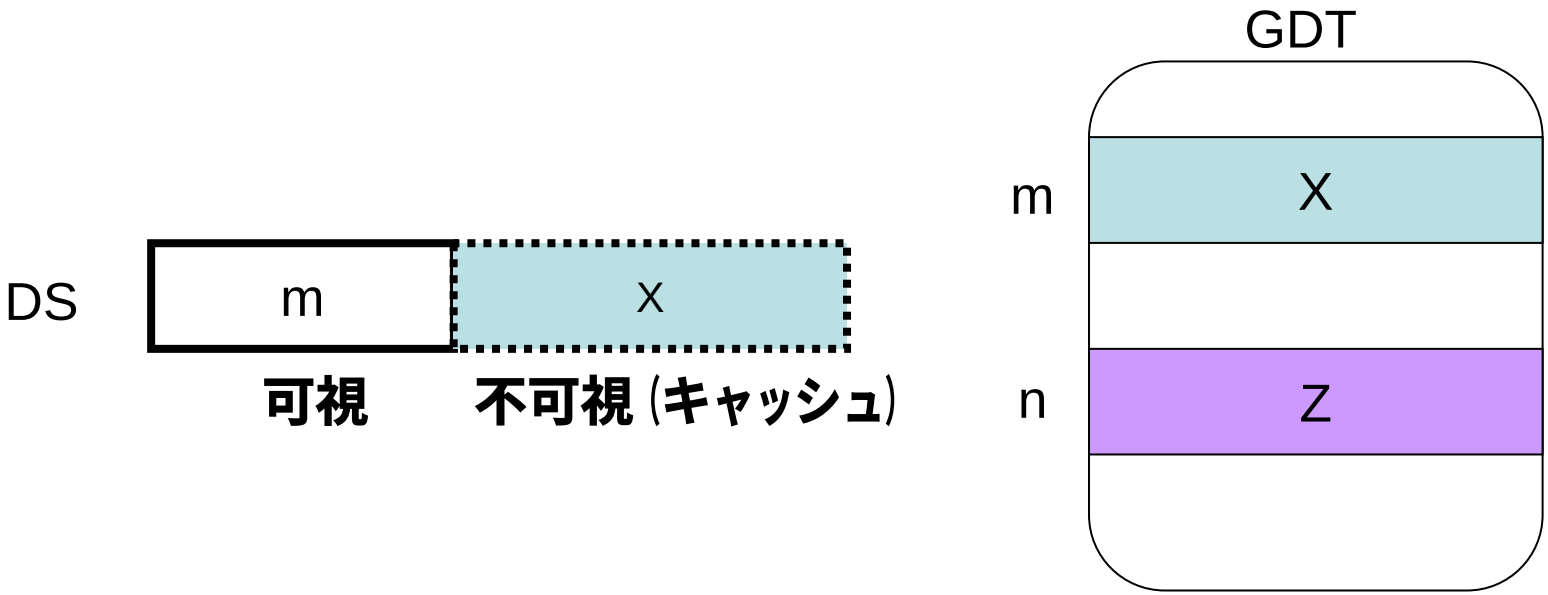
## セグメントアーキテクチャ
## アドレス変換 (ディスクリプタ書き換え)



# Intel x86アーキテクチャと仮想化における問題

## セグメントの不可逆性　ー何が問題なのか？

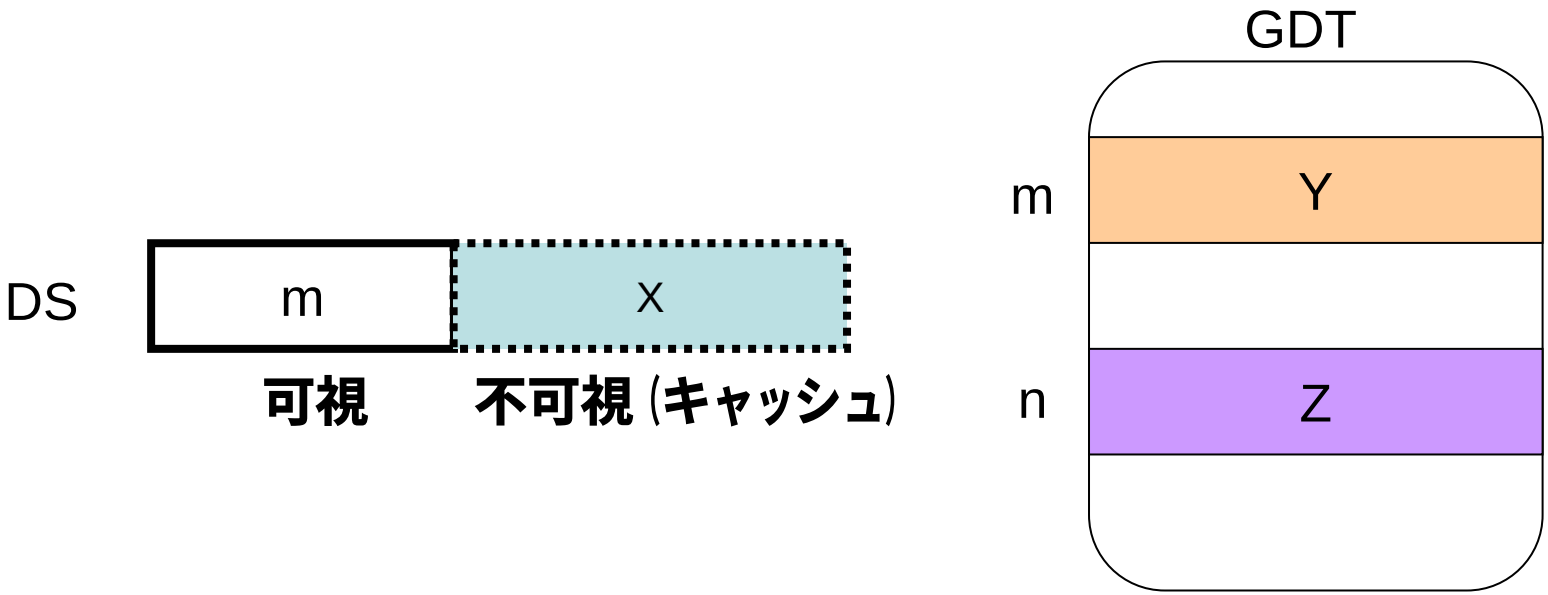
1. VMが”mov m→DS”を実行
   GDT上のオフセットmのセグメントディスクリプタの内容がセグメントレジスタDSの不可視部分にロードされる



# Intel x86アーキテクチャと仮想化における問題

## セグメントの不可逆性　ー何が問題なのか？

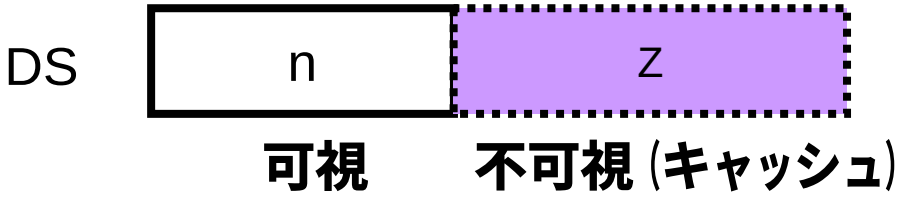
2. VMがGDTのオフセットXの内容をYに変える
   この変更はセグメントレジスタDSには反映されない

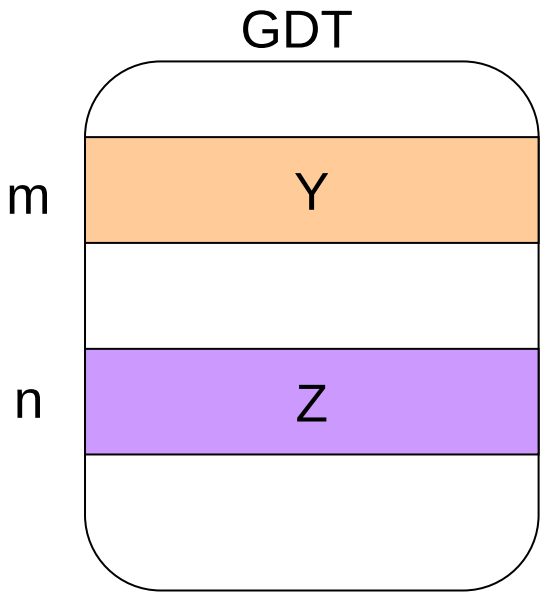


# Intel x86アーキテクチャと仮想化における問題

## セグメントの不可逆性　ー何が問題なのか？

3. VMMが一時的に(trapの処理や自身の処理で)DSを使う
   VMMが”mov n→DS”を実行
   GDT上のオフセットnのセグメントディスクリプタの内容がセグメントレジスタDSの不可視部分にロードされる
   ↓
   GDTにXの内容がないので
   DSレジスタ(X)を復元不能



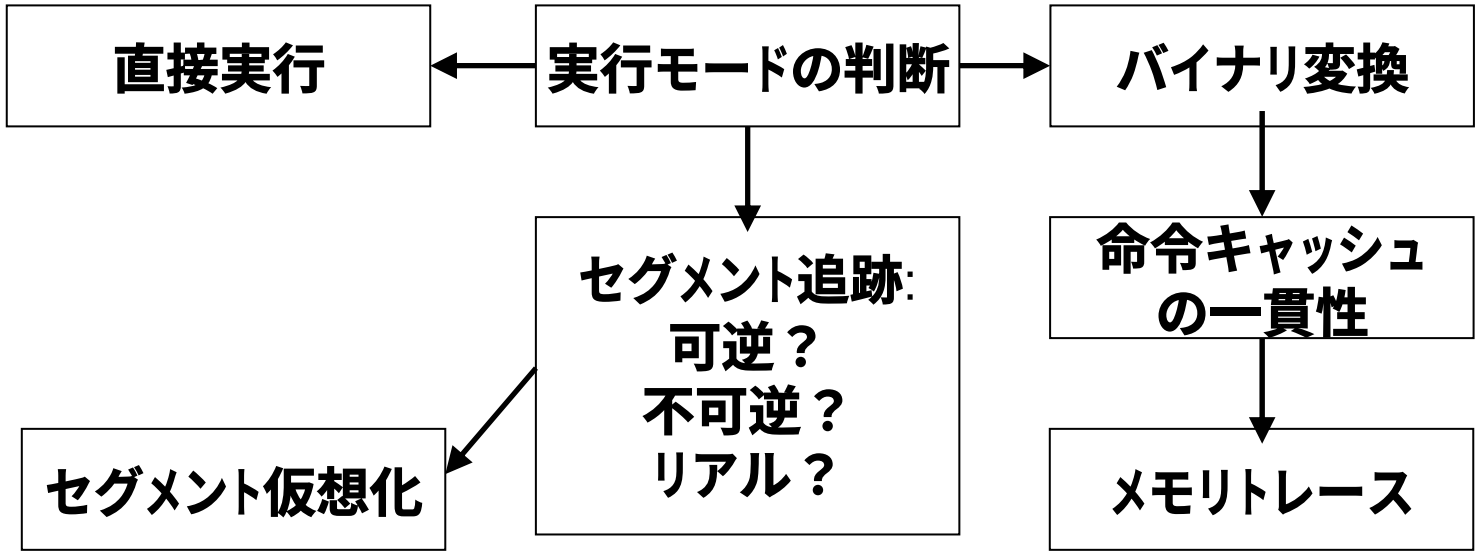
# 発明の目的

- x86アーキテクチャを高速かつ正確に仮想化する

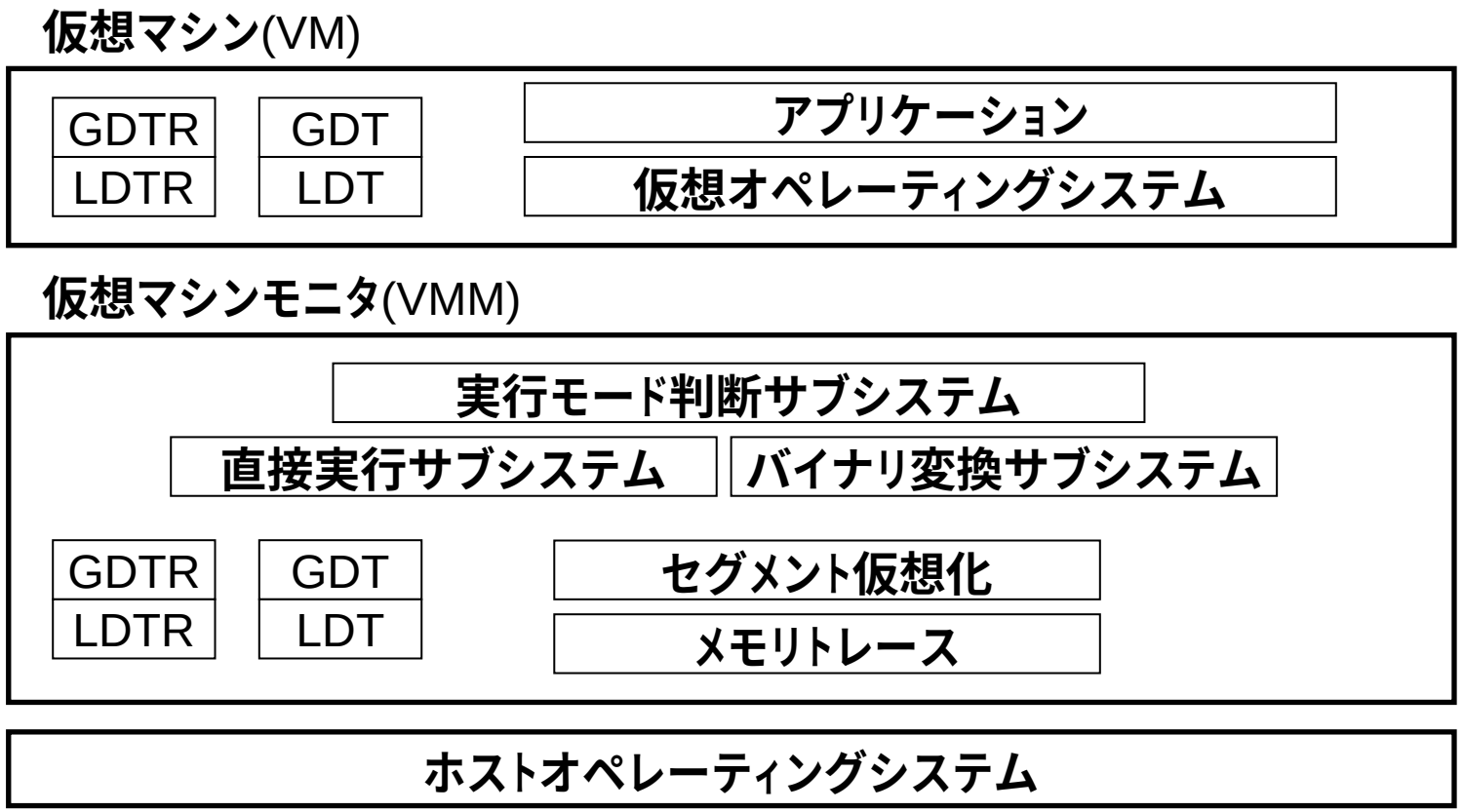
# 基本アイデア(直接実行とバイナリ変換を組合わせる)

- 速度
  プロセッサで直接実行を行う(ユーザモード)
- 仮想化不可能命令
  バイナリ変換、命令変換キャッシュ
- セグメントの仮想化
  VMMがVMのディスクリプタのコピー (シャドウ) を
  持ち、実際のセグメントレジスタに設定する
- セグメントの不可逆性
  セグメントレジスタのキャッシュをエミュレート
  (VMMの中にセグメント不可視部分のコピーを持つ)
- ページトレース
  VMの中のテーブル、被変換コードを監視

# 発明の概要


直接実行
実行モードの判断
バイナリ変換
セグメント追跡:
可逆？
不可逆？
リアル？
命令キャッシュ
の一貫性
セグメント仮想化
メモリトレース

# 発明の概要 ーシステムアーキテクチャ



# 各機能の説明

- 実行モードの判断
- 直接実行サブシステム
- バイナリ変換サブシステム
- セグメントの仮想化 (プロテクトモード)
- メモリトレース (プロテクトモード)

# 実行モードの判断

x86の三つの処理モード

- v8086モード
  完全仮想化可能→直接実行

- リアルモード、システム管理モード
  保護ができないので仮想化不可能→バイナリ変換

- プロテクトモード (一番のターゲット)
  - ユーザモード
    ユーザが特権情報を読めるのは問題なしと判断→直接実行
    ただし不可逆セグメントが有る状態ではバイナリ変換
  - 特権モード
    バイナリ変換 (不可逆セグメント有り、無し)

# 直接実行サブシステム

・直接実行とは?
ハードウェアプロセッサがVMのコードを直接実行する

VMMはVMを非特権モードで実行することで、VMによって
VMMが破壊されることを防ぐ

↓

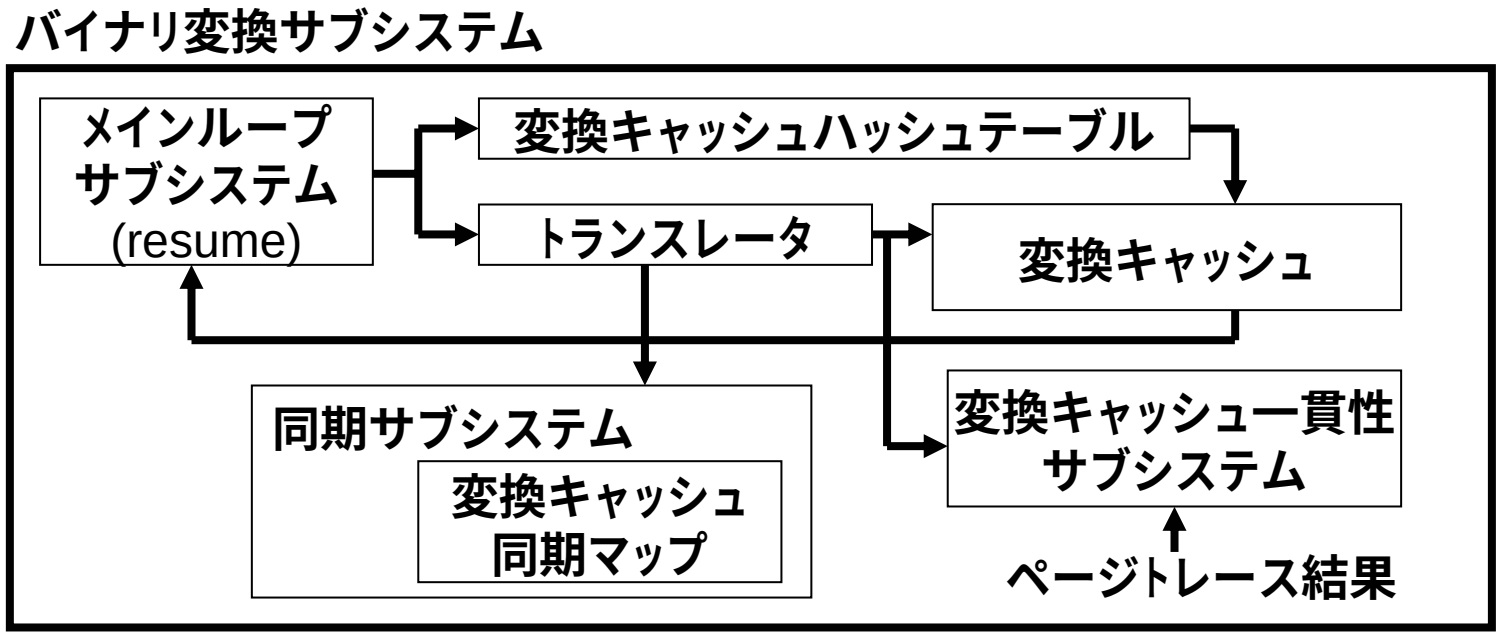
しかし、Intel x86アーキテクチャは仮想化不可能命令を
持つので仮想特権モードを非特権モードで直接実行する
ことはできない。

↓

結局、仮想ユーザ (非特権) モードのみが非特権モードで
直接実行される

# バイナリ変換サブシステム

**ハードウェアプロセッサが直接実行を行えない状態にあるときには、バイナリ変換サブシステムがVMの実行を行う責任がある。**



# バイナリ変換サブシステム

- 主な機能
  バイナリ変換モードの時に命令を変換し、変換キャッシュに結果を置き、実行する。つまり、仮想特権モードではVM上の命令列は実行されない。

- 他の機能
  - 再利用性
    オリジナルコードが変更されるまでは変換結果を再利用
  - チェイニング
    ー命令ずつではなく、命令列で変換してキャッシュする
  - アトミック性
    ー命令を変換すると複数命令になる場合があるが、この複数命令はアトミック性を持たなければならない。

# バイナリ変換サブシステム

・メインループサブシステム
　(バイナリ変換時のメインループ)

　次々と命令をエミュレートさせるメインループ

# バイナリ変換サブシステム

・トランスレータ

仮想マシンから命令列を読み、対応する命令列を生成する。
特権命令が含まれていた場合には、特権命令をエミュレートする命令列を生成する。

↓

変換結果は変換キャッシュへ

また、変換キャッシュと元の命令列の対応を、変換キャッシュ同期マップに書き込む

# バイナリ変換サブシステム

・変換キャッシュ
  変換された命令列が置かれる巨大なバッファ

・変換キャッシュハッシュテーブル
  仮想マシン上の命令列先頭アドレスと、変換キャッシュ上の
  命令列先頭アドレスの対応を保持している

・変換キャッシュ同期マップ
  仮想マシンのIP(Instruction Pointer)をシミュレートするため。
  変換キャッシュに対して命令列の長さごとの情報を持つ
  ↓
  エラーが起こった場合に元のIPを計算する際に使用する

# バイナリ変換サブシステム

・命令キャッシュ一貫性サブシステム

あるページにある命令列を変換して、変換キャッシュに保持している場合、元のページの命令列が変更されたかどうかを検出する必要がある。

↓

ページトレース機能を利用して、書き込みが発生した場合にtrapできるようにしておく

↓

書き込みが発生した場合、そのページに含まれていた命令列に対するキャッシュをすべて除去する
(単純であるが効果的な方法)

# セグメントの仮想化

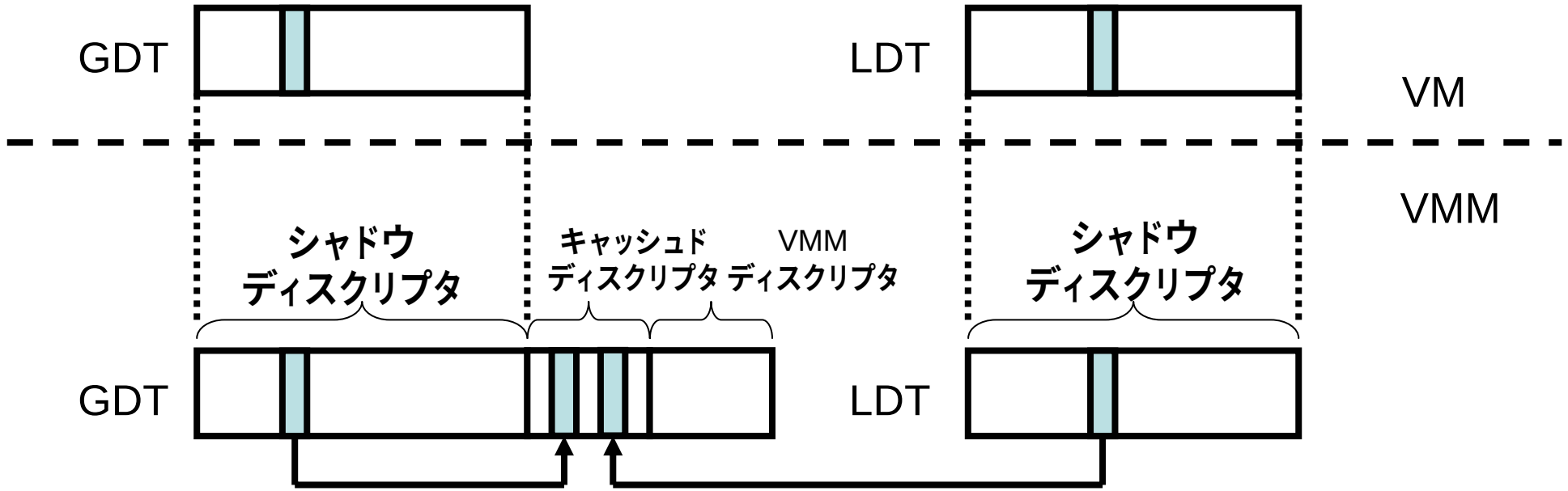
- VMMはセグメントを仮想化するために複数のディスクリプタテーブルを持つ

- VMMはセグメントを仮想化するためにハードウェアプロセッサのGDTRをVMMのGDTにセットする。VMのテーブルを直接ハードウェアプロセッサが読むことはない。

- VMMのディスクリプタテーブルとVMのディスクリプタテーブルの一貫性を保証する

# セグメントの仮想化

・VMMのディスクリプタテーブル



# セグメントの仮想化

- ディスクリプタのシャドウコピー
- セグメントキャッシュのエミュレート
- ディスクリプタ変換
  - 直接実行時のディスクリプタ変換
  - バイナリ変換時のディスクリプタ変換
- シャドウコピーの一貫性制御
  - (セグメントトレース)

# セグメントの仮想化

- ディスクリプタのシャドウコピー (シャドウディスクリプタ)
  VMのGDT/LDT内のディスクリプタをコピーし、変更を追跡する

- セグメントキャッシュのエミュレート (キャッシュドディスクリプタ)
  プロセッサのセグメントキャッシュをエミュレートする
  6つのセグメントレジスタに対応して6エントリ持つ

- VMMディスクリプタ
  VMM内部利用のためのディスクリプタ

# セグメントの仮想化

・ディスクリプタ変換 (プロテクトモード)

VMのディスクリプタテーブルとVMMのシャドウディスクリプタテーブル(ハードウェアが参照している) は一貫性を保つ

↓

しかし、一致している必要はない

データとコードディスクリプタエントリに対しては、シャドウエントリはVMMのエントリのコピーであるが以下の例外がある:

1)VMMとVMはリニアアドレスを共有しているため、
ぶつかっていた場合にはVMのリニアアドレスを切り詰める
→これが原因の一般保護例外は正しくエミュレートする

2)ディスクリプタ特権レベルが0である仮想特権モードの
ディスクリプタは1にしてバイナリ変換を行う。

3)ゲートディスクリプタのエントリは0にする (VMMがエミュレート)

# セグメントの仮想化

・直接実行時のディスクリプタ変換

VMがVMMのセグメントディスクリプタを直接読んでしまうと困る（セグメントディスクリプタはVMM内で共有）

↓

直接実行におけるVMの特権レベル(CPL)の最小は2である

↓

そこでVMM内のディスクリプタのディスクリプタ特権レベル(DPL)はすべて2未満(0 or 1)にしておく。

VMがディスクリプタをロードしようとするとtrapが発生する

# セグメントの仮想化

・直接実行時のディスクリプタ変換

VMM内のディスクリプタのディスクリプタ特権レベル(DPL)はすべて2未満(0 or 1)になっている。

↓

VMが非シャドウディスクリプタ（ディスクリプタテーブルのサイズの問題で全部はシャドウできない）をロードしようとすると一般保護例外になる。

↓

バイナリ変換モードにしてエミュレートする

# セグメントの仮想化

・バイナリ変換時のディスクリプタ変換

バイナリ変換時はVMとVMMは同じ特権レベル(=1)で動作する。

↓

したがってセグメント特権レベルを利用して非シャドウのディスクリプタのロードを防ぐことはできない。

↓

セグメントのロードもバイナリエミュレーションする。

# セグメントの仮想化

・シャドウコピーの一貫性制御（セグメントトレース）

VMがGDT/LDTをLGDT/LLDT命令を利用して
切り替えたときには、この特権命令をエミュレートする

・すべてのシャドウエントリを再変換する
・新しいGDT/LDTにページトレーを設定する

# セグメントの仮想化

・シャドウコピーの一貫性制御 (セグメントトレース)

VMがGDT/LDTのディスクリプタに書き込みを行った場合の処理
(GDT/LDT) にはページトレースを設定してあるのでtrap可能

・現在セグメントレジスタにロードされているディスクリプタか？
no→
そのディスクリプタをシャドウディスクリプタにコピーする
(ディスクリプタ特権レベルも下げる)
yes→
・VMMにあるGDT/LDTエントリの古い値をキャッシュド
エントリにコピーする (セグメントキャッシュのエミュレート)
・シャドウディスクリプタを更新する
・バイナリ変換モードにする

# メモリトレース

・しくみ

MMUの機能を利用して物理ページに対して、読み出しトレース、
または書き込みトレースを設定する。

ページフォールトが発生するので、トレースを解除して、
シングルステップ実行し、再びトレースを設定する。

トレースされたVMには透過であり、検知されない

・逆マップ
リニアアドレスを指定してトレースを行ないたい場合、トレースを
おこなっている物理ページに対する仮想ページのリストを保存
する

# メモリトレース

・制限
MMUを使用するのでページ単位になってしまう。

・メモリトレースを使う理由

1)命令変換キャッシュとオリジナルコードの一貫性確保
2)ページテーブルベースのMMUを仮想化する
3)GDT/LDTとシャドウディスクリプタの一貫性確保
↑
すべて対象ページをトレースすることで実現する

# 本発明によるシステムの例

・一つのハードウェア上で異なるＯＳのアプリケーションを
　動作させることができる。

Windowsアプリケーション + Unixアプリケーション
(WindowsとUnixのどちらがホストＯＳでも良い)

・ホストOSのない実装も可能

・マルチプロセッサ環境における実装
　・複数のVMの並行実行
　・一つのVMに複数の仮想プロセッサ
　　この場合、メモリトレースの拡張が必要
　　(トレース事象が発生した際にプロセッサ間の対話が必要)