

平成 25 年度

筑波大学情報学群情報科学類

卒業研究論文

題目
Offloading device drivers into a hypervisor
for separation of concerns

主専攻 情報システム主専攻

著者 米司 伊織

指導教員 加藤和彦

要 旨

オペレーティングシステムは様々な機能を提供しているが、その中でデバイスドライバの占める開発コストは大きい。そこで本論文では、デバイスドライバを提供する低オーバーヘッドのハイパーバイザと、よりシンプルなオペレーティングシステムからなる新しいオペレーティングシステムアーキテクチャ・デザインを提案し、オペレーティングシステムの開発における「関心の分離」を図る。その主要な技法として Device Masquerading について論じ、本デザインの応用について検討する。また、その評価に際して実際に BitVisor を改変して実装し、性能について実験する。また、さらにネットワーク性能を改善させる方法について議論し評価する。

keywords = BitVisor, Hypervisor, OS Design, libraryOS, embedded computing, OS noise, open standard, virtio, paravirtual device

目次

第 1 章	序論	1
1.1	既存のデバイスドライバの問題点	1
1.2	提案	2
第 2 章	設計と技法	3
2.1	Device Masquerading Underlay でのドライバの定義	3
2.2	Device Masquerading	4
2.3	Zero-copy Device Masquerading	7
第 3 章	実装	8
3.1	Device Masquerading Underlay の技術的仕様	8
3.1.1	Device Masquerading Underlay の隠蔽	8
3.1.2	virtio デバイスの実装上の技法	9
3.1.3	適切なデバイスドライバの検索とハンドリング関数への関連付け	10
3.2	EPT によるフックの動作	12
3.3	virtio の仕様について	14
3.3.1	PCI configuration 空間について	14
3.3.2	virtio の capability list について	16
3.3.3	PCI configuration 空間の改変	17
3.4	virtqueue の仕様について	17
3.5	virtio-net の仕様について	20
3.5.1	送信時の動作	21
3.5.2	受信時の動作	21
第 4 章	評価	22
4.1	Intel 1GbE でのスループット	22
4.2	Intel 1GbE でのレイテンシ	23
4.3	セキュリティ	26
第 5 章	結論	27
5.1	まとめ	27

5.2	今後の発展	28
5.2.1	対応デバイスクラスの追加について	28
	ブロッククラス	28
	グラフィックスについて	28
	Memory ballooning クラス	28
5.2.2	Function Kernel の組み合わせについて	29
5.2.3	Exitless Device Masquerading	29
付録 A	実験環境について	30
A.1	Intel PRO/1000 を用いた実験環境について	30
付録 B	ソースコード	31
付録 C	PCI configuration 空間の例	32
C.1	Intel PRO/1000	32
C.2	virtio-net デバイス	33
参考文献	35

目次

2.1	既存のハイパーバイザ (ここでは Xen) でのデバイスアクセス	5
2.2	parapass-through hypervisor でのデバイスアクセス	5
2.3	提案手法と伝統的なオペレーティングシステムを組み合わせたデバイスアクセス . .	6
2.4	提案手法と Unikernel を組み合わせたデバイスアクセス	6
2.5	Device Underlay Architechture	6
2.6	Zerocopy Device Underlay Architechture	7
3.1	メモリマップ	9
3.2	ページディレクトリ方式のページテーブルでのアドレス変換の概略図	12
3.3	ページテーブルと EPT を組み合わせたアドレス変換の概略図	13
3.4	Capability list	16
3.5	virtqueue の 1 バッファのライフサイクル	18
3.6	リングバッファの構造	19
4.1	DMU がある場合とない場合、及びある場合のキューサイズごとのスループットの測定	23
4.2	DMU がある場合とない場合と KVM 上のゲストとの通信の場合、及びある場合のキューサイズごとのレイテンシの測定	24
4.3	(左) PRO/1000 に対する Linux の挙動 (baremetal)	25
4.4	(右) PRO/1000 に対する Device Masquerading Underlay と Function Kernel の挙動	25

序論

1.1 既存のデバイスドライバの問題点

近年、オペレーティングシステムのアーキテクチャは転換期をむかえている。その要因として、モバイル・IoT の普及とともに、オペレーティングシステムの低消費電力化と同時に多様なデバイスに対応しなければならなくなっているほか、ハイパフォーマンスコンピューティング環境の巨大化によって、よりオペレーティングシステムへの性能要求は高まっている。しかしながら、同時にオペレーティングシステムの開発コストは年々上がっている。その理由として、多様な SoC やデバイスの普及によって、デバイスドライバの開発コストは年々上がっているという点が挙げられる。デバイスドライバは歴史的にオペレーティングシステムの不具合の原因の最たるもので [3]、近年改善されているとはいえ、頻度・件数ともに主要因の一つでありつづけている [7]。また、オペレーティングシステムの開発コストもデバイスドライバは支配的な存在となっている。このことを示すためにコード行数ベースで比較すると、一例として linux の mainline(1 月 13 日時点での最新) 全体に対するデバイスドライバの占める割合は 1884 万行中 1171 万行で 62.2% となっている。この問題はデバイスドライバが各オペレーティングシステム間で再利用できないことによる。

また、ライブラリ OS の研究がここ数年で盛んになっていて実際に民用においても使われ始めているが、多くのライブラリ OS の実行環境は伝統的なオペレーティングシステムの上に限定されていて、ライブラリ OS のパフォーマンスが引き出せるとは限らない。ライブラリ OS ではカーネルをライブラリとしてリンクし、実行ファイルでありながら同時にオペレーティングシステムとして動作するもので、仮想化環境の中で動作させることを前提に、仮想ハードウェアの性能を引き出すことを目的にしているが、もしこれが実際のハードウェアを占有して動作することができれば、より高い性能改善が期待できる。また、コンパイルされてリンクされたライブラリ OS のライブラリとアプリケーションは、様々な利点が指摘されているものの、同時に、メンテナンスするためにコンパイルしなければならないという問題点があげられる。負荷の変動によって実行環境をライブマイグレーション [?] などを用いて移すことは出来るものの、通常の OS のように直接ハードウェアの上で実行するためにはそれぞれのライブラリにドライバを導入し再リンクしなければならないという困難があり、現実的ではない。

さらに、オペレーティングシステムは検証の難しい巨大なシステムソフトウェアであるので、これへの attack surface は非常に大きい。そのためオペレーティングシステムそのものが攻撃者の

意図するプログラムを実行してしまう危険性が高く、オペレーティングシステムがデバイスへ直接アクセスできる場合にはデバイスを破壊したり、デバイスを乗っ取って攻撃者にとって都合がよい動作をさせる可能性がある。例えば、キーボード入力を取得し送信するプログラムを隠蔽して動作したり [10]、バッテリーのファームウェアに侵入し発火・爆発させることや [5]、起動できなくさせること [?] が可能となる。

1.2 提案

上記の問題はデバイスドライバの共有化が進んでおらず、それぞれのオペレーティングシステムでデバイスドライバの開発を別々に行わなければならないという点や、それぞれのデバイスがそれぞれのアクセスメソッドを持っていて、統一化されていないという点によって生じている。そこでデバイスドライバをハイパーバイザにオフロードすることで、オペレーティングシステムの開発という、様々な機能の実装を連携して進めなければならなかったものからデバイスドライバの開発を分離し関心の分離を図る。また、この分離によって、仮にオペレーティングシステムの脆弱性を利用して任意の特権コードが実行できるようになっても実デバイスの代わりに仮想デバイスを提供していることによって実際の有効な破壊を未然に防ぐことができるようになる。

そのために本論文では Device Masquerading という技法とそれを利用したハイパーバイザを提案し、これを利用することでデバイスとシステムソフトウェアの関係を包括的に捉えなおす新しいオペレーティングシステムアーキテクチャについて議論する。

具体的には、オペレーティングシステムの機能を、デバイスドライバのようなハードウェアとのインターフェース部分と、ファイルシステムやプロセス管理などのソフトウェア部分を分離し、前者を Device Masquerading Underlay, 後者を Function Kernel と呼ぶ。この二つは virtio と呼ばれる共有メモリのリングバッファの標準規格を通して通信する。

Device Masquerading Underlay は Function Kernel に対して非依存であるため、これが提供するデバイスドライバを複数のオペレーティングシステムが利用できるようになる。また、virtio という標準規格を用いるので既存のオペレーティングシステムがそのまま Function Kernel として利用できる。この仕様に則ったアクセスメソッド以外の操作を必要としないので、複数のインスタンスを起動させることが主目的であるハイパーバイザで必要になる煩雑な設定が必要ない。また attack surface を極力小さく抑えることができる。

さらに、この実装を Parapass-through Hypervisor である BitVisor を元に行い、性能評価を行った。その結果、レイヤが増えているにも関わらず、性能はほとんど変わらず、むしろ改善する場合もあった。とくに応答速度は、virtio デバイスを提供する既存の Type II 型 VM である KVM は遅くなるのに対し、Device Masquerading Underlay はむしろ速くなるということがあきらかになった。

設計と技法

2.1 Device Masquerading Underlay でのドライバの定義

デバイスドライバの再利用については、Le Vasseur らの論文 [4] で提案された手法が興味深い。これもデバイスドライバの再利用を仮想化技術を用いて行うことで、ターゲットにするオペレーティングシステムにデバイスドライバを搭載せず、オペレーティングシステム非依存の形でデバイスドライバを利用できるというものだ。この論文では、ターゲット環境はホスト環境にあり、デバイスドライバを提供する層がその上でゲスト環境として動作するというアーキテクチャが提案されている。この論文でも仮想化とメモリ機構を利用し十分に高速に動作するほか、この論文での利点として、ゲスト環境でドライバを提供するオペレーティングシステムは既存のものを拡張したものなので、既に市場で使われているデバイスドライバが利用できるという点があげられる。しかし、デバイスドライバとしてオペレーティングシステムをゲスト環境で動作させることによるフットプリントが大きくなることが予想されるほか、ドライバを提供されるオペレーティングシステムが仮想化をサポートしないとならない。

ユーザのオペレーティングシステムに制限や新たな必要要件を与えずにデバイスドライバを提供するには、ユーザのオペレーティングシステムよりも先に起動したほうがよい。そこで、デバイスドライバを提供するソフトウェアをハイパーバイザとして実装し、ユーザのオペレーティングシステムをその後起動することにした。このソフトウェアを Device Masquerading Underlay と呼び、後者のユーザのオペレーティングシステムを便宜的に Function Kernel と呼ぶことにする。

Device Masquerading Underlay は Function Kernel に対して非依存なデバイスドライバを提供するが、同時に自身がデバイスドライバを提供しないデバイスについては Function Kernel からの操作を許したい。これによって、Device Masquerading Underlay が提供できるデバイスがあまり多くない状況から本研究で提案するアーキテクチャを採用することが出来、スムーズな移行が可能になる。

そこで Device Masquerading Underlay と Function Kernel のアクセスするそれぞれのデバイスについて考えることにする。

実際にマシンに存在するハードウェアの集合を Dev_{real} 、Device Masquerading を施す機能を関数 $M(x)$ と表記するとする 2.1。

また、 M の定義域を $Dev_{hyp_can_drive}$ とかくとする 2.2。

$$M(x) := x \text{を} DeviceMasquerade \text{する} \quad (2.1)$$

$$M : Dev_{hyp_can_drive} \vdash Dev_{kernel} \quad (2.2)$$

すると、Function Kernel が発見できるデバイスのリストは以下のようになる 2.3。

$$Dev_{kernel} = map(M, Dev_{real} \times Dev_{hyp_can_drive}) \oplus (Dev_{real} \setminus Dev_{hyp_can_drive}) \quad (2.3)$$

このように、Device Masquerading Underlay に対応するドライバがないデバイスは Function Kernel に直接渡される。このことを実現するために、 Dev_{kernel} と Dev_{real} はどちらもデバイスの集合でなければならない。ということは、関数 M はデバイスを受け取ってデバイスを返す関数となる。通常のデバイスドライバはデバイスを操作してソフトウェアのインターフェースを提供するが、Device Masquerading Underlay はデバイスを操作しつつ、デバイスのように振る舞う必要がある。

通常のオペレーティングシステムに内蔵されたデバイスドライバは、オペレーティングシステムの他の機能と関数呼び出しを用いて通信するので、オペレーティングシステムの実装や ABI に依存してしまうというだけでなく、この手法では Device Masquerading Underlay の機能を実装できないことになる。そこで本研究では、Device Masquerading Underlay をハイパーバイザとして実装し、Function Kernel は仮想化されたゲスト環境の上で動くとし、Device Masquerading Underlay は一般の仮想化技術を用いて仮想化されたデバイスを提供することとする。この時、この Underlay がドライバを持っていないデバイスはそのままゲスト環境に提供するようにするため、Parapass-through [9] という技術を用いる。

Device Masquerading Underlay はハイパーバイザとして実装されるので、Function Kernel が起動する前に Device Masquerading Underlay が起動する必要がある。そして、自身がデバイスドライバを持っているデバイスのみ初期化し、後述する Device Masquerading を施す。そして残りのデバイスとともにこれらをゲストに提供するために、parapass-through 技術を用いる。

2.2 Device Masquerading

Parapass-through Hypervisor は一般のハイパーバイザとは異なり、ほとんどのデバイスをゲスト OS に提供する。一般のハイパーバイザは、管理者 OS の上で動作し、ゲスト OS には自身のデバイスを提供しない。例えば Xen では Dom0 が実際のデバイスを動作させ、DomU の環境でデバイスが必要になった場合には、DomU の OS に含まれた専用デバイスドライバが XenBus などを通して仮想デバイスあるいは準仮想化デバイスを動作させ、Dom0 がこれによって必要になる I/O をエミュレートし再現する 2.1。また、QEMU+KVM においても、仮想デバイスか準仮想化デバイスをホストのプロセスとして動作する QEMU がエミュレートし、カーネルがゲスト

OS をアプリケーションとして処理し、管理する。一方、Parapass-Through Hypervisor の実装である BitVisor ではホスト OS を含むアービテータが存在せず、ただ一つのゲスト OS のみが常に動作する。ハイパーバイザは介入が必要な例外が発生した場合に限り最小限動作し、必要がない限り、実在するデバイスへのアクセスをゲスト OS に許している。これにより不必要なエミュレーションコストを排除でき、ハイパーバイザによる余分なコストが非常に小さくなっている [9]。

Device Masquerading とは、実デバイスへのアクセスをハイパーバイザがゲスト OS に提供許可する代わりに、ハイパーバイザ自身が実デバイスのアクセスを行い、リクエストされた情報を高速なリングバッファである virtio[8] を通してゲスト OS と通信するというものだ。

ここで、既存のハイパーバイザによる準仮想化と本研究で提案する Device Masquerading との違いを説明する。

上述のように、既存のハイパーバイザではアクセスに要するパスが多い上に管理用のオペレーティングシステムが必要になる 2.1。それに対し、bitvisor のような parapass-through hypervisor ではデバイスを pass-through することができるうえ、デバイスへのアクセスの途中で介入しデータを改変することが出来るが、実デバイスのドライバがゲスト OS に必要だった。

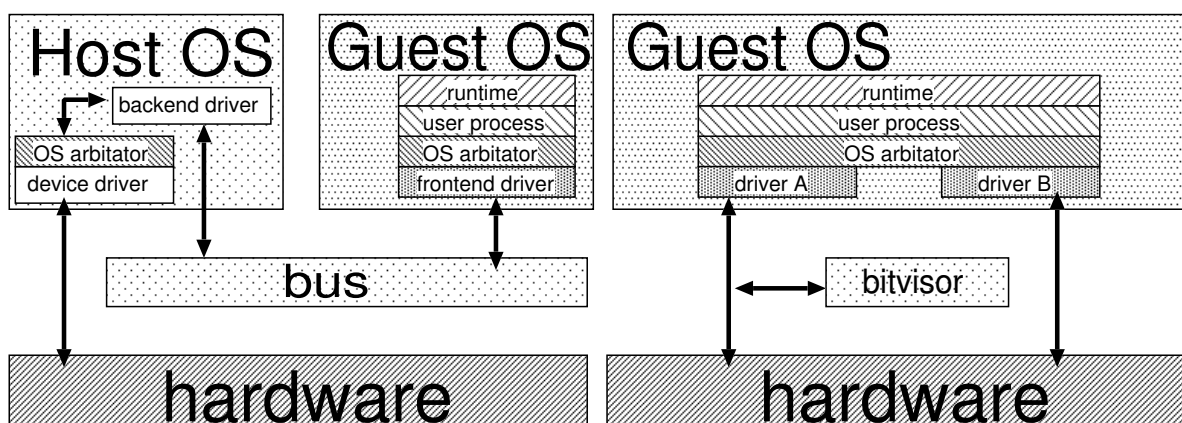


図 2.1 既存のハイパーバイザ (ここでは Xen) でのデバイスアクセス

図 2.2 parapass-through hypervisor でのデバイスアクセス

それに対し、Device Masquerading を行う小さなハイパーバイザ (ここでは Device Masquerading Underlay と呼ぶ) を導入すると、ゲスト OS (ここでは Function Kernel と呼ぶ) には実デバイスの制御を行う必要性がなく 2.3、オペレーティングシステムに求められる他の機能の開発に注力できる。Function Kernel はそれぞれのデバイスクラスに対してただ一つのデバイスドライバさえあればよく、この一つのドライバをすべてのユーザが使ううえ、このドライバは実デバイスを扱わないため、ドライバのカバレッジが高く堅牢になる。

Device Masquerading Underlay はオペレーティングシステム非依存なため、より多くのユーザが使用でき、より検査されるほか、より多くの開発リソースが期待でき、またそのメリットは複数のオペレーティングシステム開発者に還元されることになる。

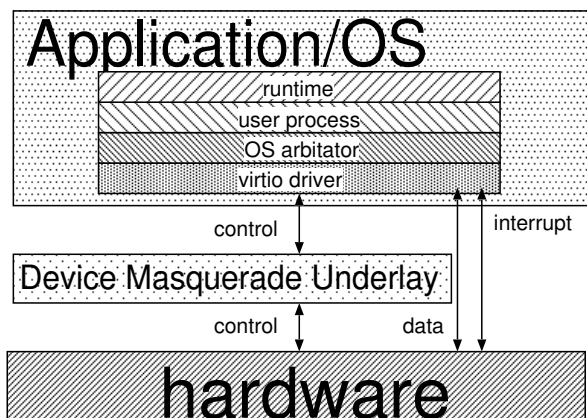


図 2.3 提案手法と伝統的なオペレーティングシステムを組み合わせたデバイスアクセス

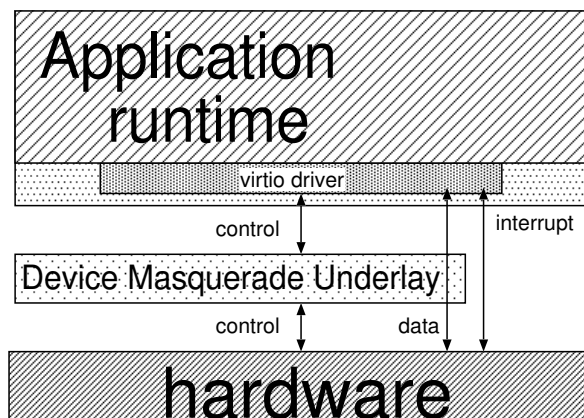


図 2.4 提案手法と Unikernel を組み合わせたデバイスアクセス

具体的な実装として、おもに Device Masquerading Underlay は 2 つの部分からなる。第一に、実際のデバイスを制御したり、デバイスとやりとりする部分である。第二に、virtio デバイスとして振る舞う部分である。

virtio デバイスとして振る舞う部分は共通のコードを使うことができ、デバイス固有のドライバ部分のみをそれぞれ作成すればよい。この二つのコミュニケーションは Device Masquerading Underlay の内部のバッファを用いて行う。

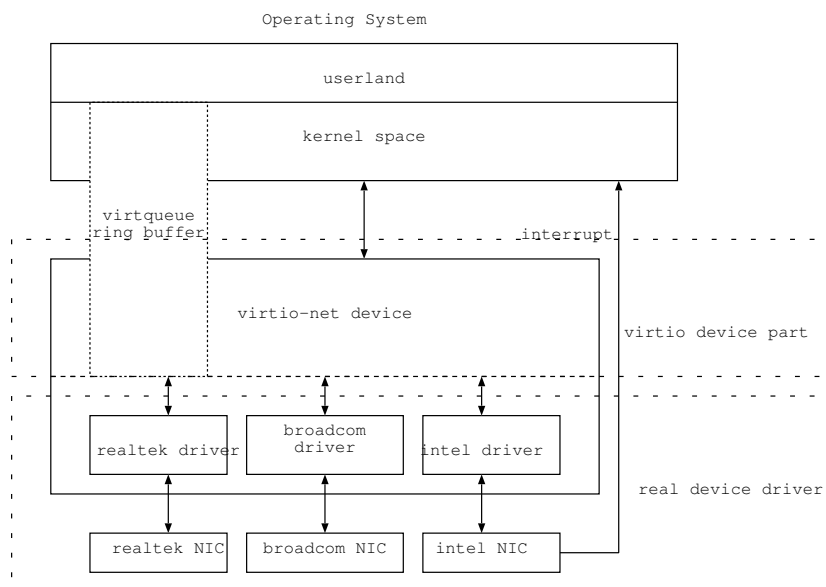


図 2.5 Device Underlay Architecture

2.3 Zero-copy Device Masquerading

Device Masquerading Underlay は実際のデバイスを制御しつつ、virtio というデバイスがそこに実在しているかのように見せることで実現とするが、これだけでは性能的な困難が解消できない。そこで本章では、オーバーヘッドの解消のための技法を提案する。

まず大きな性能改善の妨げとして、メモリコピーの速度の問題があげられる。オペレーティングシステムは多くの場合、通信の内容を書き込んだメモリの in-kernel buffer をリングバッファとしてデバイスに提供することで高速に通信しているが、図 2.3 のようなアーキテクチャでは、virtio デバイスとして振る舞う箇所と個別のデバイスドライバの箇所でも同様にバッファを用いてやり取りする。これは不要なコピーが余計に増えていることになる。もっとも遅い箇所は後述のようにデバイスへのコミュニケーションだが、virtio 化の部分に限るともっとも遅い箇所はメモリコピーであった。

そこでメモリコピーを減らし、不可欠なデバイスへのコピーを除いてメモリコピーを最終的にはゼロにしたい。この手法を独自に Zero-copy Device Masquerading と名付け実装する。図で示すと図 2.6 のようになる。

この技法の概要は、主に Function Kernel から与えられたメモリバッファをそのままデバイスへ書くというものだが、そのためには個別のデバイスドライバが virtio 特有の処理を行わなければならない、図 2.3 で表したような分離が難しくなる。そこで個別のデバイスドライバでの virtio 対応の手間を極力減らすために実装上の工夫が必要となった。

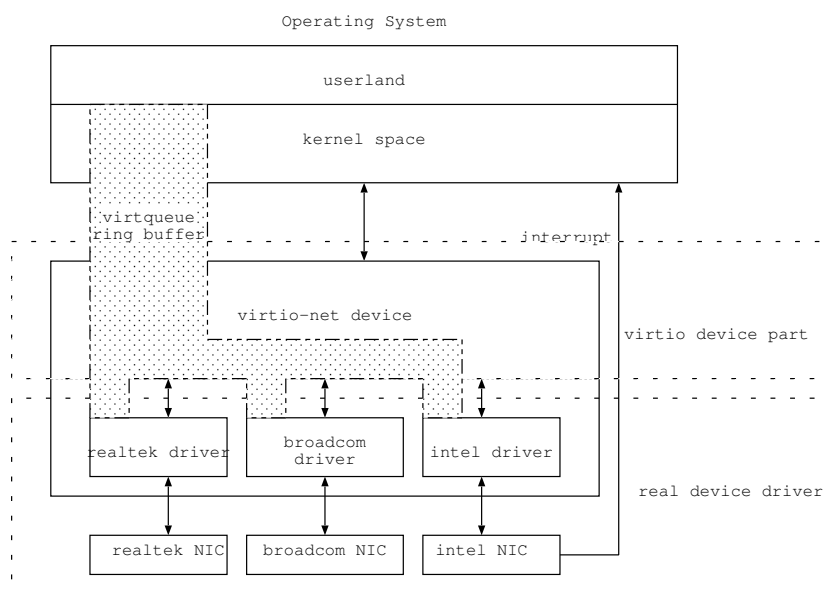


図 2.6 Zerocopy Device Underlay Architecture

実装

3.1 Device Masquerading Underlay の技術的仕様

Device Masquerading Underlay の仕様は上記のとおりだが、実装上の制約をいくつか述べる。まず前述のように必須な処理として OS より先に起動するという点が挙げられる。ここで行う処理は主に、Device Masquerading Underlay のプログラム領域の保護や、virtio デバイスの作成、それから virtio 化するターゲットとなるデバイスの検索と初期化である。

3.1.1 Device Masquerading Underlay の隠蔽

まず自らの隠蔽について述べる。

Function Kernel はどの領域に Device Masquerading Underlay があるのか調べる方法がないので、メモリ領域全体に書き込みを行いかねない。また、自らが管理しているはずのメモリ領域に書いた場合、正しく読み出せなければならない。であるから、Device Masquerading Underlay はそもそも Function Kernel が書き込まず、また万一書いたとしても書いたとおり正しく読み出せなくてもよい特別な領域を作って「潜む」必要がある。そのような領域として ACPI テーブルの末尾が挙げられる。

ACPI テーブルとは、電源管理やデバイスツリーの管理のために用意されたメモリ領域で、プラットフォームに非依存の標準規格で定められたものである。ACPI テーブルはファームウェアによって提供されるが、この領域も Device Masquerading Underlay と同じく上書きされてはならないメモリ領域で、だが Device Masquerading Underlay とは異なり、自らの領域を OS に通達することができる。そこで BitVisor では ACPI テーブルの領域を取得後、その領域に続く場所に自らのプログラムをコピーしなおす。このように起動中の処理として自らのプログラムを再配置する処理は既存のオペレーティングシステムでも一般的な技法である。しかし BitVisor がこれらとは異なるのは ACPI テーブルの直後の箇所に再配置するという場所の差異の他に、ACPI 領域を取得するオペレーティングシステムの操作をインターセプトして、自らの領域を含めた「より大きい」領域がまるで ACPI テーブル領域として予約されているかのように振る舞うという点となる。このように確保したメモリ領域にオペレーティングシステムが読み書きしようとした場合は無効になり、常に 0 が読み出されるようになっている。このことを可能にするのは SPT または

EPT という技術である。この技術については 3.2 章で解説する。

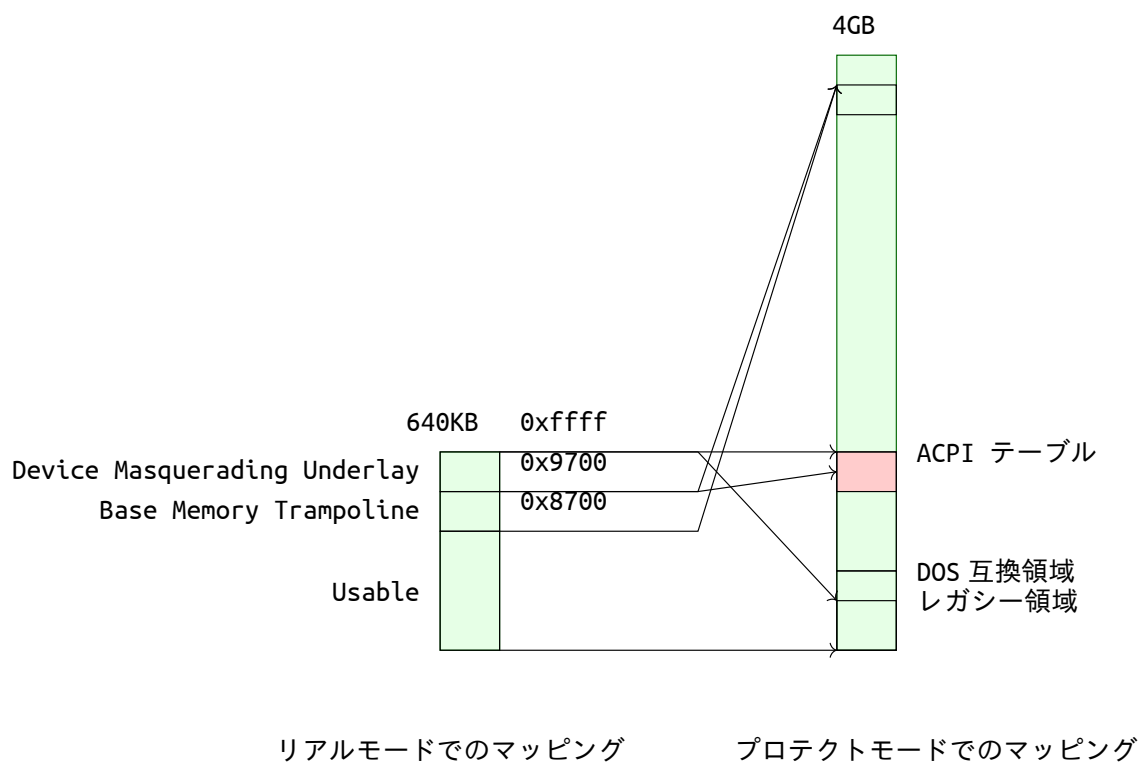


図 3.1 メモリマップ

Device Masquerading Underlay も BitVisor の技法を踏襲し、EPT およびメモリマップの構造を取得する命令のインターセプトを用いて Function Kernel からの破壊を防ぎ、セキュリティを担保している。

3.1.2 virtio デバイスの実装上の技法

一般のハイパーバイザは virtio デバイスをゲストに提供するに当たって新しく作り、virtio デバイスとして振る舞うエミュレータがアプリケーションとして動作するため、それぞれのハイパーバイザが動作するオペレーティングシステムでのネットワーク通信用のブリッジなどと通信すればよい。しかし本論文での実装では、一つの実在するハードウェアに相当する virtio デバイスを作成し関連付けるので、より効率的な作成方法が考えられる。

この方法を順を追って説明する。

1. 後述のように Masquerading の対象となるデバイスを発見する
2. その PCI configuration 空間に virtio の PCI configuration 空間として有効なものとなるようフック関数を用意する (PCI configuration 空間については 3.3.1 章で後述する)
3. Function Kernel は BAR と呼ばれる領域のビットマスクを最初に読み出すので適切な値

をフック関数で返す

4. のちにそれぞれのデバイスに相当するデバイスドライバを割り振るために Vendor ID と Device ID を読み出す必要があるので、これを virtio のものにする
5. Function Kernel はそれぞれのデバイスに相当するドライバが確定すると、その固有のドライバをアサインするので virtio の規格に則り各フィールドを適切な値として返す

これによって、新たにデバイスの領域を作りだすかわりに、デバイスの種類をすり替えることが出来るようになる。このことで、実際のデバイスが存在するバスの位置に virtio デバイスが存在するように見せかけられるので、抽象化の行き過ぎによる混乱をさけることが出来る。また、新たにデバイスを作成するには PCI configuration 空間を発見するための ACPI テーブルごと変える必要があるが、これは専用の中間言語で操作される高度に複雑なもので、このことでコードサイズが無用に大きくなってしまい attack surface が拡大するが、このすり替えによる virtio デバイスの作成によって簡単に実際のデバイスの隠蔽も出来、コードサイズ小さくなって処理が簡単になるため、よりよい。

3.1.3 適切なデバイスドライバの検索とハンドリング関数への関連付け

デバイスの探索には ACPI テーブルを読み出して PCI configuration 空間を発見し、ターゲットとなるデバイスを示す ID を持っているものを見つけないといけない。適切な ID とそれに合った初期化ルーチンの関連付けは `struct pci_driver` と呼ばれる構造体として記述し、この構造体をリストに登録する上位の初期化ルーチン `void pci_register_driver(struct pci_driver)` を用いて登録する。この上位の初期化ルーチンは、前述したデバイス固有の初期化ルーチンとは異なって、実在するデバイスの数のみならず対応しているデバイスのドライバの数だけ呼び出される必要があるが、小さな linked list への登録をそれぞれ一度だけ行うものなのでこの実行のコストは無視できる。

`void pci_register_driver(struct pci_driver *)` を実行する関数は `PCI_DRIVER_INIT(X)` と呼ばれるマクロを通して登録される。BitVisor ではいくつかの `_INIT` で終わる名前のマクロが存在し、ここで与えられた関数は、このマクロの優先度の順に実行される。`PCI_DRIVER_INIT` マクロは PCI デバイスの初期化よりも前に実行され、ここで呼ばれた `void pci_register_driver(struct pci_driver *)` によって登録された `struct pci_driver` 型のグローバル変数は ACPI テーブルを BitVisor が解釈した際に探索され、この構造体のメンバ関数である `.new` が呼ばれるためにある。このメンバ関数として与えられた初期化ルーチンには、引数としてデバイスオプションが与えられ、このオプションはデバイスの種別やバスの位置などに基づいてユーザがオプションとして与えるものである。

ここでマッチした構造体のメンバ関数は、与えられたオプションや PCI デバイスの固有の情報を与える `struct pci_device` という構造体を用いて初期化を行い、上述のような処理を行うフック関数などを登録して終了する。

理解のためにソースコードの一部分を示す。

```
1 static void
2 bnx_new (struct pci_device *pci_device)
3 {
4     :
5     :
6     :
7 }
8
9 static struct pci_driver bnx_driver = {
10     .name      = driver_name,
11     .longname  = driver_longname,
12     .driver_options = "tty,net,virtio",
13     .device    = "class_code=020000,id="
14         "14e4:165f|" /* BCM5720 */
15         "14e4:165a|" /* BCM5722 */
16         "14e4:1682|" /* Thunderbolt - BCM57762 */
17         "14e4:1684|" /* BCM5764M */
18         "14e4:1686|" /* BCM57766 */
19         "14e4:1691|" /* BCM57788 */
20         "14e4:16b4", /* BCM57765 */
21     .new      = bnx_new,
22     .config_read  = bnx_config_read,
23     .config_write = bnx_config_write,
24 };
25
26 static void
27 bnx_init (void)
28 {
29     pci_register_driver (&bnx_driver);
30 }
31
32 PCI_DRIVER_INIT (bnx_init);
```

ここで、PCI_DRIVER_INITマクロによって `void bnx_init(void)` が呼ばれることが約束される。これが呼ばれると `struct pci_driver bnx_driver` が登録される。もし BitVisor が起動時にこ

の構造体の `.device` で示される文字列に相当する Vendor ID 及びデバイス ID のいずれかを持っている PCI デバイスを発見した場合は、その場で `.new` で示された `void bnx_new (struct pci_device *pci_device)` を呼ぶ。この中でデバイスを初期化したり、前述のフック関数を登録したりすることになる。

3.2 EPT によるフックの動作

EPT とは、Extended Page Table の略称で、Second Level Address Translation という技術の Intel 製品での実装である。まず読者のために、ページテーブルについて軽く説明する。

近代的なアプリケーションは実際のメモリの上で直接動作する代わりに、固有のメモリアドレス空間の上で動作している。しかし固有のメモリ空間を作り出しつつ実際に動作するためには、そのアドレス空間と実際のメモリ空間との間に関連付ける方法が必要になる。その一つとしてセグメント方式が挙げられる。しかし、より高度な分離を行う必要が出てきたのでその後実装された変換方式がページテーブルである。

ページテーブルは 32bit あるいは 64bit のアドレスを、下位の 12bit と残りの 10bit ずつのキーに分割する。それぞれのキーのうちもっとも上位のものをインデックスとしてアドレス変換テーブルを引き、それで得たテーブルに対し残りの最も上位のキーをインデックスとして用いてさらに引き、引き終わったものに先ほどの 12bit を append してアドレスを得る。

この場合最初に引くテーブルをプロセスごとにあてがえば、プロセスごとに独立した空間を作ることができるため良いアドレス変換が出来る。この最初に引くテーブルを指定するものが CR3 レジスタと言われる特殊なレジスタである。

現在使われているオペレーティングシステムのほぼすべてはこの仕組みを利用してアドレス変換を行ってプロセスを独立させている。

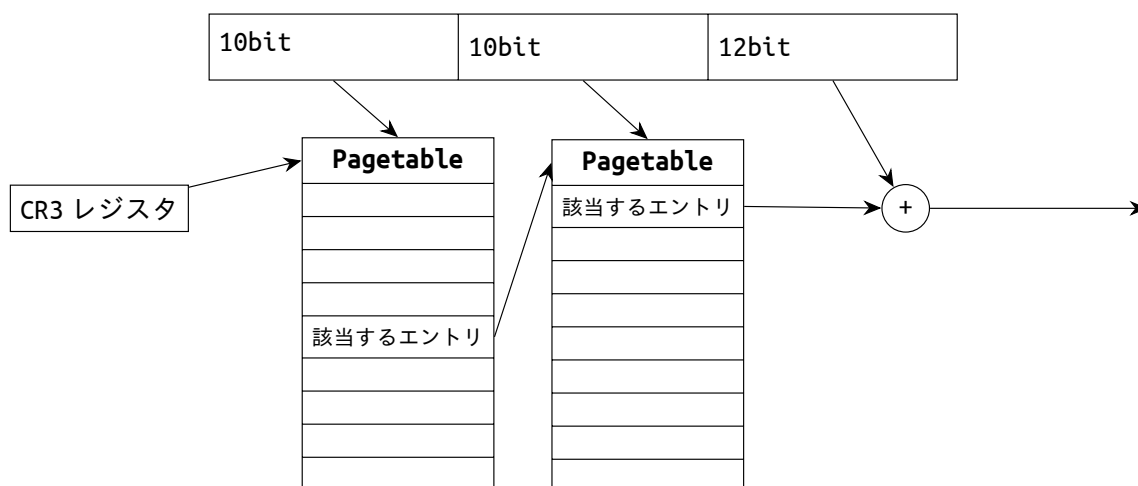


図 3.2 ページディレクトリ方式のページテーブルでのアドレス変換の概略図

このようなアドレス変換のおもな目的として、プロセス空間の分離やセキュリティが挙げられ

る。プロセスのアドレス空間と物理的なメモリ空間を別のものとして扱うことで、プラットフォーム全体へのアクセスを限定することができる。例えば、リングプロテクションと併用することで、プロセスは CR3 を書き換えられず、プロセス同士の侵入やプラットフォームへの直接のアクセスを防ぐことが出来る。

同様の仕組みを仮想化支援機構として導入したのが、Second Level Address Translation である。この仕組みは、上に挙げたようなアドレス変換を一段目とし、そこで得たアドレスをさらに変換するというものだ。

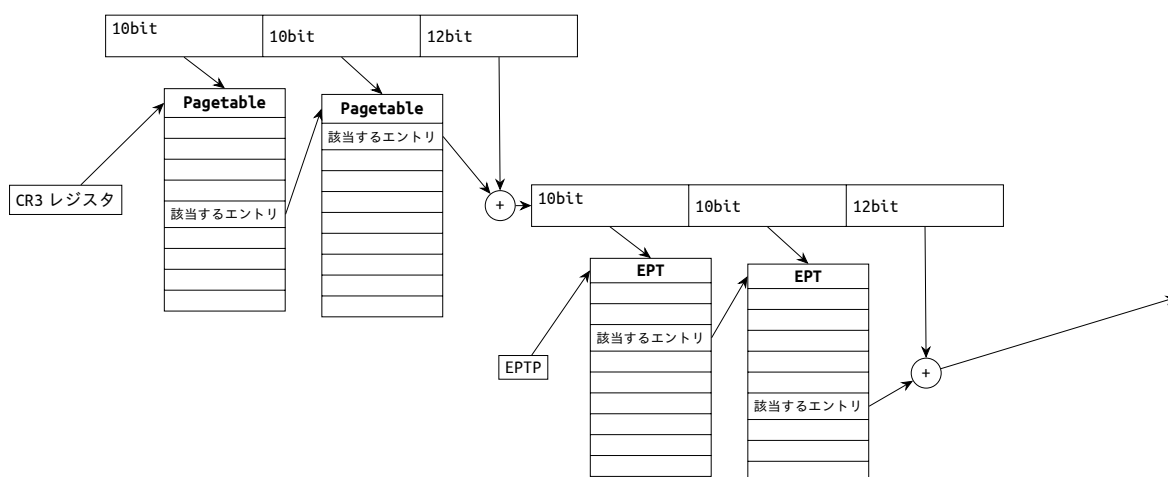


図 3.3 ページテーブルと EPT を組み合わせたアドレス変換の概略図

これを行うことで、仮想化ゲストのオペレーティングシステムはハイパーバイザを配慮してメモリを配置したり、あるいはハイパーバイザがゲストのオペレーティングシステムの CR3 を監視して、プロセスが代わるたびに CR3 をセットしたりする必要がなくなった。

ゲストは一般のハードウェアの上で動作しているという前提でアドレス空間を利用してよいし、またそのアドレス空間が物理的にどこにマップされるかはハイパーバイザが決定することが出来るようになった。

Intel の仮想化支援機構では、VMCS と呼ばれる仮想化ゲストが発行出来る命令やその他の様々な制御用構造体が用意されていて、これをセットアップすることで仮想マシンを作ることが出来る。EPT によるアドレス変換の際のエラーハンドリングや、IO 命令発行時のハンドリングなどをする際には、それらが終了理由になるようにし、`vmlaunch` 及び `vmresume` 命令により、セットアップした VMCS を実際にゲストとして実行してからそのコンテキストが終了するまで待つと、こうしたイベントをフック出来るようになる。

EPT やページテーブルのエントリには、次のテーブル (ディレクトリ) へのポインタの他に制御用のビットフィールドがあり、これによってページのサイズやエラー時に起きるべき例外の選択などが出来る。

3.3 virtio の仕様について

virtio は準仮想化デバイスのデファクトスタンダードとして広く用いられており、ライブラリ OS のライブラリでもサポートされる数少ないデバイスである。この仕様は一度大きく変わっている。今回は既に普及している古い規格ではなく、性能改善のため新しい virtio 1.0 と呼ばれる規格に準拠する。以降の説明は virtio 1.0[6] での定義にしたがう。

この規格では、デバイスをその動作の種別によっていくつかのデバイスクラスに分けている。最もよく知られたものに virtio-net と呼ばれるネットワーククラスがある。今回はこれを proof-of-concept の実装として対象にした。このクラスは最もよく知られ、広く利用されているだけでなく、パフォーマンスの測定に便利だという特徴がある。その他のデバイスクラスとして、コンソールやブロックデバイス、ビデオカードなどが挙げられる。

デバイスは起動時の検索によって発見されなければならないが、virtio 自身はその方法を定めていない。virtio デバイスは、以下の 3 つのバス規格のうち 1 つを利用することになっている。

- ・ PCI
- ・ MMIO
- ・ Channel-IO

このうち、Channel-IO は IBM の専用のハードウェアを前提にしたもので、今回は実装の対象ではない。また、MMIO の場合はそもそもバスというよりは PCI のようなデバイス検索の機能を持つハードウェアを利用できない組み込みプラットフォームなどを対象にしたもので、MMIO 空間のどこにマップされるか予め管理者が入力しておくというものだ。これは不便で、PCI が使える場合には使われず、あまり知られていないので今回は実装の対象としない。

今回は PCI バスでドライブされる virtio デバイスを作成することとした。これは最もよく知られた規格で、ACPI によって PCI バスは列挙されているのでデバイス検索が可能であり、Function Kernel の可移植性を引き上げることが出来る。

一般にこの場合 virtio デバイスとして振る舞う PCI configuration 空間を作成する必要があるが、本研究で提案するアーキテクチャの場合、新たにデバイスを作り出すのではなく既存のデバイスを Function Kernel からみて virtio デバイスとして振る舞わせ、利用させるので、既存のデバイスの PCI コンフィギュレーション空間を活用すれば良い。

3.3.1 PCI configuration 空間について

このサブセクションでは、デバイスを使うゲスト上のソフトウェアをドライバ、virtio デバイスを含む PCI デバイスをデバイスと呼ぶ。

PCI configuration 空間とは、PCI デバイス固有のアドレス空間で、0x00 番地から 0xff 番地までが利用できる。このうち特に 0x00 番地から 0x3f 番地まではそれぞれのフィールドの意味が決まっている。

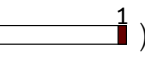




ここで、特に説明に必要なフィールドについて先に列挙する。

- Vendor ID
- Device ID
- BAR[0..5]
- Subsystem Vendor ID
- Subsystem ID
- Capability pointer

ここで、BAR とは、3.1.2 章ですでに触れた BAR のことである。VendorID と Device ID は適切なデバイスドライバを発見するために必要なもので、デバイスの製品ごとに固有のものだ。これについてもすでに触れた。Subsystem Vendor ID や Subsystem ID も Vendor ID や Device ID と同じ機能を持つ。

Capability とは、PCI configuration 空間で予め定められたフィールド以外の設定をデバイスが通知したい時に、PCI configuration 空間内に linked list を作ることになっているが、そのリストの head を指す番地が入るのが Capability pointer である。

BAR はドライバとデバイス双方が読み書きする 32bit の領域だ。ここでアドレスなどがやり取りされることが多い。BAR は Base Address Register の略で、ここにかかれた値が具体的に何に使われるのかは定められていないが、メモリ上のなんらかの構造体へのポインタをやり取りして、その構造体を用いてデバイスとドライバがコミュニケーションすることが多い。virtio も同様に BAR に virtio 固有の設定をするための構造体を示すアドレスがドライバからかけられることになっている。このような使い方をする際、単純にどのようなアドレスがデバイスに支給されても必ず使えるとは限らない。PCI はプラットフォームに非依存の規格であるから、例えばバイトアドレッシングに対応していないデバイスで、下位 2bit が 00 でなければ動作しないデバイスや、キャッシュを跨いでいると使えないデバイス、あるいは DMA の都合上一定のアドレス領域でなければならないということがありえる。このような場合に備え、BAR へ書くことができるアドレスを調べる方法が用意されている。BAR に書けないビットは何を書いても同じ内容しか読み出せないはずである。ということは、まず全ビットを 0 にして書き込み、読み出してからまた全ビットを 1 にして書き込んで読み出して比べると、書き込めるビットが分かる。

また、BAR の下位 2bit あるいは 4bit には意味がある。下位 1 bit () が 1 であった場合、その BAR にかかれた値は PIO 空間へのポインタとなる。その場合、2 bit () めは予約領域である。下位 1 bit () が 0 であった場合、その BAR にかかれた値はメモリ空間へのポインタとなる。その場合、そこから数えて上位 2bit () はアドレスの大きさを指し、その次のビット () はプリフェッチ可能か示すビットである。

ここで指定されたアドレスには、一体どのような構造体ができるのだろうか。実は virtio では、ここで交換すべき情報として 5 つの構造体が定義されている。BAR には 6 つのエントリがあるが、

これを対応させるのは capability list で行う。

3.3.2 virtio の capability list について

これら 5 つの構造体の種別と BAR 番号をひもづける capability list の構造は virtio の規格で決まっている。

そもそも capability list とは以下のようなもので、特にその capability のエレメントがどのような意味をもつのかを表す ID と 次のエレメントへのポインタはどの capability であっても必ず書いてあり、それぞれのオフセットは 0 と 1 である。リストの最後の要素は、次のエレメントへのポインタとして 0 と書いてなければならない。

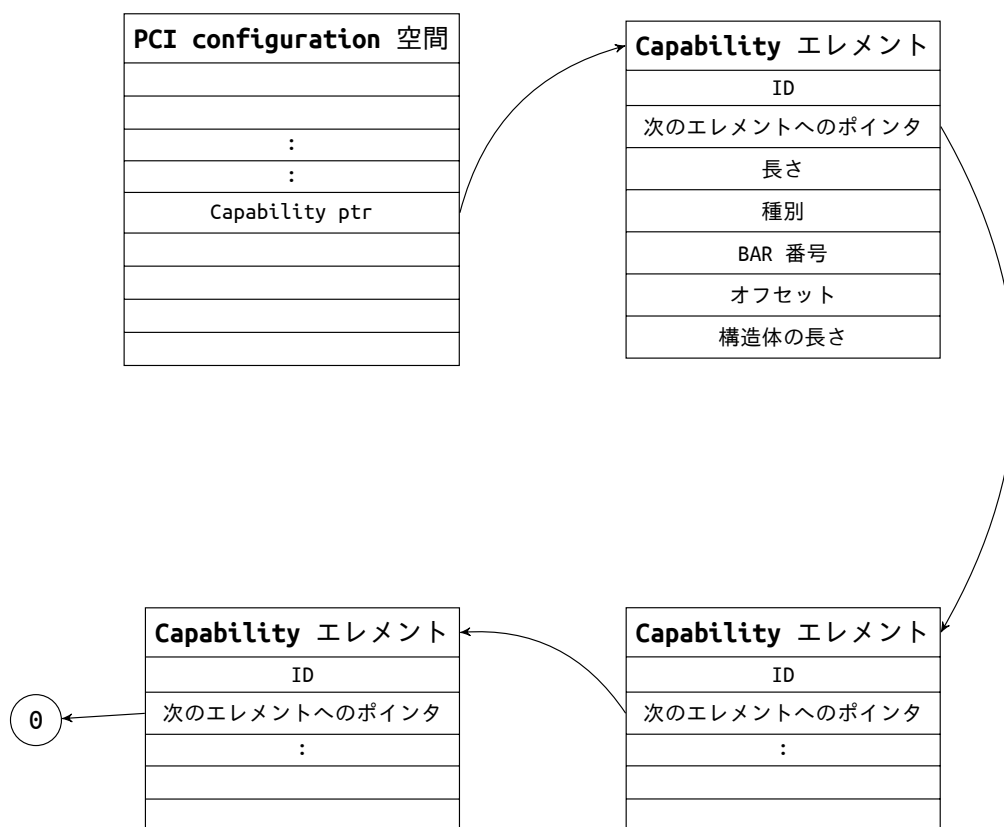


図 3.4 Capability list

ここで、最初の **capability エレメント** は、virtio の capability list の形をしている。長さはこのエレメントの長さ、種別は構造体の種別を表している。

実際にこのような capability list を作るのは簡単で、このリストも PCI configuration 空間内のものだから、PCI configuration 空間と同様に改変すれば良い。

ただし、このエレメントはどの内容だろうと自在に配置されてしまうので、MSI 割り込みを有効にする場合には困難が伴う。というのは、MSI 割り込みの capability list は MSI 割り込みを

利用する場合には必須だが、このフィールドの内容はデバイスの実際の内容を反映しなければならないし、ドライバが提供するアドレス情報やデータ情報をデバイスに書き戻さなければならない。こうした情報を BitVisor で予め読み書きし、ドライバから書き込まれた場合にはその内容を書き戻す必要がある。これは、デバイスに提供した MSI 割り込みの capability list の位置とは異なる可能性があるうえ、このエレメントに示される ``次のエレメントへのポインタ`` は無効な値であるから適宜読み替えなければならない。さらにその上、こうした capability list の要素が多いデバイスは、PCI configuration 空間とはアクセスの方法が全く異なる Extended PCI configuration 空間にも情報が書いてある可能性があるので、MSI 割り込みを必ず有効にするということは大変だ。

3.3.3 PCI configuration 空間の改変

PCI configuration 空間の改変には、Function Kernel からの PCI configuration 空間の読み書きをインターセプトして改変すれば良い。PCI configuration 空間への読み書きは `in/out` 命令で行うので、上記の方法でフックすれば良いことになる。起動時に読み書きするだけで、その後頻繁に読み書きするものではないから、多少のオーバーヘッドは性能に一切影響を与えない。このことはのちのベンチマークで明らかになる。

PCI configuration 空間への読み書きのインターセプトは既に先行の実装 [9] で提供される仕組みを用いると簡単に実装できる。しかし、BitVisor の多くの実装では、今回のようにたくさんのフィールドを改変するわけではないため、そのままこの仕組みを利用すると煩雑になってしまう。そこである程度宣言的に書けるよう工夫をした。

3.4 virtqueue の仕様について

virtqueue とは、virtio の仕様の根幹をなす機能である。これはメモリのリングバッファになっていて、virtio を用いた Function Kernel と Device Masquerading Underlay の通信の多くはこのリングバッファを経由して行う。virtqueue のメモリはドライバの責任で用意する。デバイスはゲストが用意したメモリを読み書きし、使い終わったら返却する。この仕組みを利用して デバイス \longleftrightarrow ドライバ 間の読み書きを行う。

virtqueue のリングバッファのエレメントは任意の大きさのバッファである。このバッファのライフサイクルを示す。

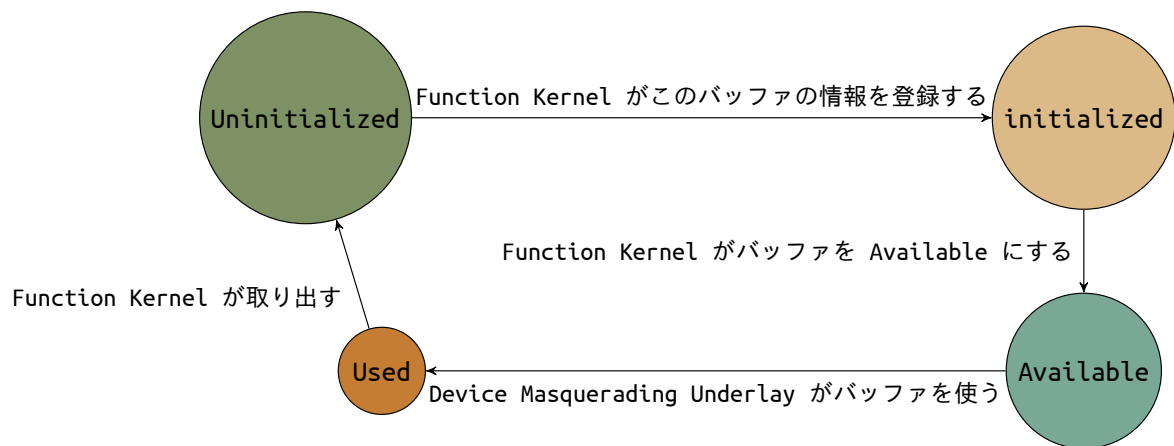


図 3.5 virtqueue の 1 バッファのライフサイクル

また、これらのバッファを複数管理するためにリングバッファになっているので、そのリングバッファの構造を示す。リングバッファは、**used** と **avail** と呼ばれるカウンタによってその一部を指されている。**avail** が **used** に追い抜かれないという制約を付けると、必ずデバイスにとって **available** なバッファは $used \leq x < avail$ の間にあるということが分かる。さらに、リングバッファに直接データを書くのではなく、リングバッファがデータをやり取りする場所を指し示すようにすることで、リングバッファ全体の大きさを一定に保つほうが良い。

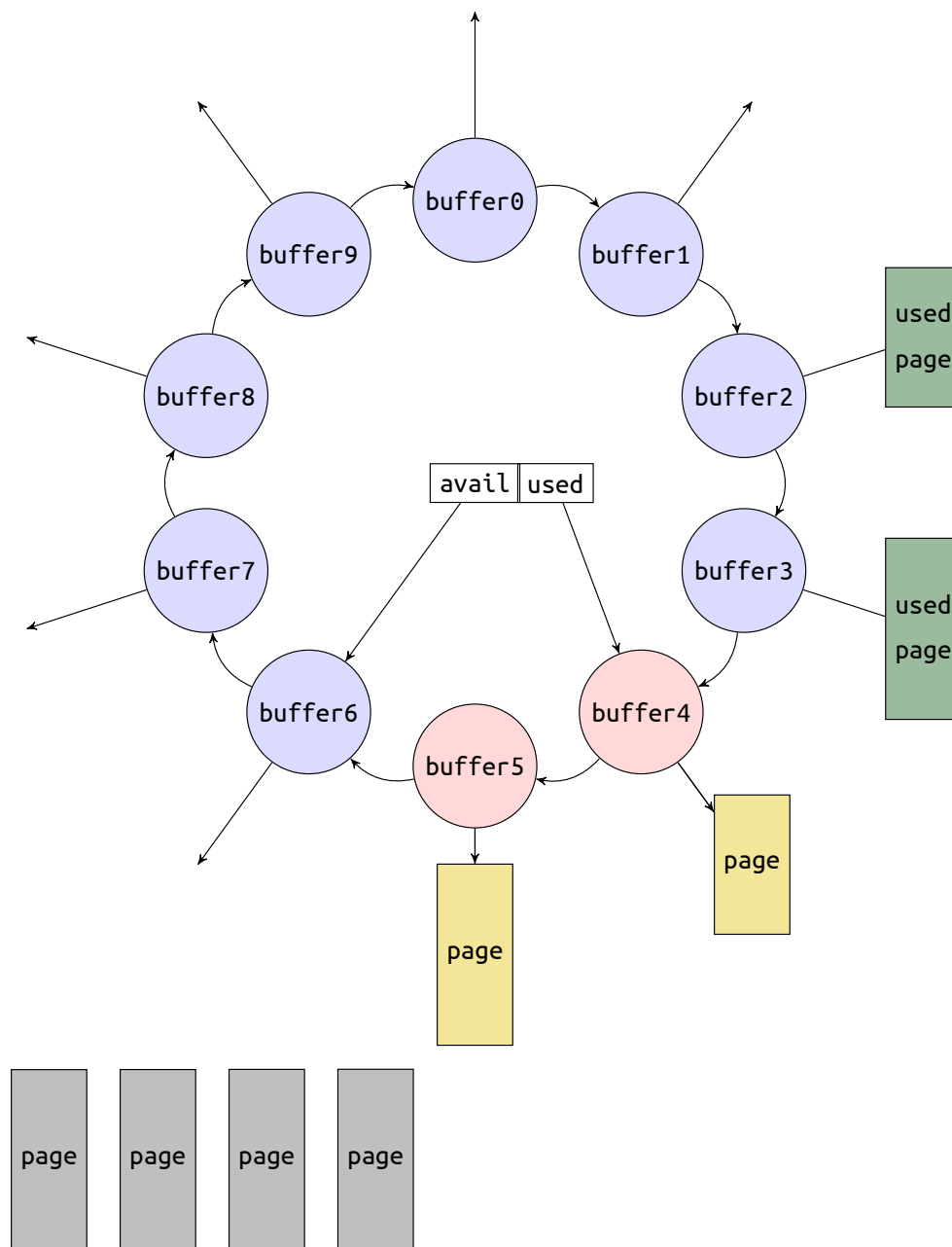


図 3.6 リングバッファの構造

このリングバッファを作るために必要なデータについて考えると、**avail** と **used** のポインタだけでなく、データをやり取りする **page** の状態や隣のリングバッファの番号などを保持する各リングバッファの要素のディスクリプタを作った方が良い。実際 virtqueue はこの通りに作られている。

virtqueue を構成する具体的な要素について列挙する。

- ・ ディスクリプタ

- Available Ring
- Used Ring

以上のエントリが連続しているメモリ領域を Function Kernel が作成し、PCI 経由で通達した構造体を通して Device Masquerading Underlay に通達する。このメモリ領域の定義の抜粋を示す。

```

1 struct virtq_desc {
2     /* Address (guest-physical). */
3     le64 addr;
4     /* Length. */
5     le32 len;
6     /* This marks a buffer as continuing via the next field. */
7     #define VIRTQ_DESC_F_NEXT 1
8     /* This marks a buffer as device write-only (otherwise device read-
        only). */
9     #define VIRTQ_DESC_F_WRITE 2
10    /* This means the buffer contains a list of buffer descriptors. */
11    #define VIRTQ_DESC_F_INDIRECT 4
12    /* The flags as indicated above. */
13    le16 flags;
14    /* Next field if flags & NEXT */
15    le16 next;
16 };

```

`virtq_desc` は、個々のバッファで、図のように linked list になっている。それぞれ Function Kernel から見たバッファのアドレスが書いてあり、また有効な長さが書いてある構造体だ。`virtq_avail` は、`avail` 状態のバッファの先端の次を指しているほか、いくつかの情報を持っている。`virtq_used` は `used` 状態のバッファの次を指している。

なぜそれぞれ次のバッファを指しているかというと、もし使用済のデータを指すようであれば、初期化時の値が不明になってしまうからだ。次くる場所を指すようにすれば、当然 0 番目からスタートするので間違いがなくてすむ。

3.5 virtio-net の仕様について

`virtio-net` とは、`virtio` の標準規格 [6] で定められたデバイスクラスのひとつで、ネットワーク機能を提供する。`virtio-net` のデバイスは 3 つのキューを持つ。それぞれ、送信キュー、受信キュー、コントロールキューである。

ここでやり取りされるデータには 12 バイトのヘッダがついていて、その後にはネットワークのイーサネットフレームが続く。また、フレームが一つのバッファに入りきらないことがありえる。そのような場合は後続のデータに続いている。後続のデータは、リングバッファで示された次の要素が指し示している。Linux のドライバはヘッダとフレームを別々に送ることが多いが、特に仕様で決まっているわけではない。

3.5.1 送信時の動作

送信キューは、Function Kernel が外にデータを送りたいときに用いられ、Function Kernel がそこに書き終わると Device Masquerading Underlay に通知する。Device Masquerading Underlay はそこから取り出して実際のデバイスに割り振って送信する。

送信すべき内容は Function Kernel がまず Avail リングバッファに格納し、avail インデックスを進める。デバイスには notification 構造体へ書き込みのアクセスが行われ、このタイミングでメモリアクセスをフックし Device Masquerading Underlay が avail インデックスと used インデックスを比較、間にあるバッファを送信する。

データを送信し終わると Device Masquerading Underlay は used インデックスを進める。

Used リングバッファに格納された使用済ページは Function Kernel によって破棄される。

3.5.2 受信時の動作

受信キューは、実デバイスが Function Kernel に割り込みを通知すると、Function Kernel が Device Masquerading Underlay の割り込み原因レジスタを見にくるので、そのタイミングで実デバイスから Device Masquerading Underlay が情報を取り出し、Function Kernel に処理を戻す。

あらかじめ Function Kernel は多めに空白ページを確保しておき、avail リングバッファにつないでおく。Device Masquerading Underlay が受信処理をし、書き込み終わったものから順番に used リングバッファにつなぎ、used インデックスを進める。このとき、Function Kernel は割り込みを受けて原因レジスタを見にきたところなので、ここでパケットの受信による割り込みであると通知することでこの used リングバッファを Function Kernel が消費して受信処理が完了する。消費し終わったページはやはり Function Kernel が破棄する。適宜のタイミングで Function Kernel は avail リングバッファを補充する。この補充が遅れるとネットワーク性能に影響がでてしまう。

評価

今回実装した `virtio` \longleftrightarrow 実デバイス 変換の性能を評価するため、ネットワークの速度評価で知られる `netperf` と `ping` で実験を行った。

また、本研究での実装で、性能に影響を与える可能性があるパラメータとしてキューのサイズが挙げられる。キューが小さすぎると性能が悪化することは上記のとおりだ。そこで様々なキューのサイズで試行した。しかし、使用済のキューの廃棄は簡単であり、今回 Function Kernel として用いた Linux 4.4.0 では適切にハンドリングしているので、キューのサイズによる性能の劣化は見られなかった。また、Zero-copy 化の最適化を施しても性能は改善しなかった。メモリコピーよりもより大きなボトルネックがあることが推測される。

また、比較対象として baremetal と KVM に対して同様の実験を行った。

baremetal とは、ここでは仮想マシンをはじめとするインターセプタが存在しない、ハードウェアの上で動く既存のオペレーティングシステムの環境のことを指す。

KVM はよく知られたハイパーバイザの実装で、この実装も `virtio` を利用しているが、図 2.1 のようなアーキテクチャをしているためレイテンシが大きくなってしまう。

4.1 Intel 1GbE でのスループット

この実験は `netperf` というベンチマークソフトウェアで行った。ベンチマーク環境については、Appendix A.1 章に記した。

Virtio で用いる リングバッファのキューのサイズを変えて実験を行った。

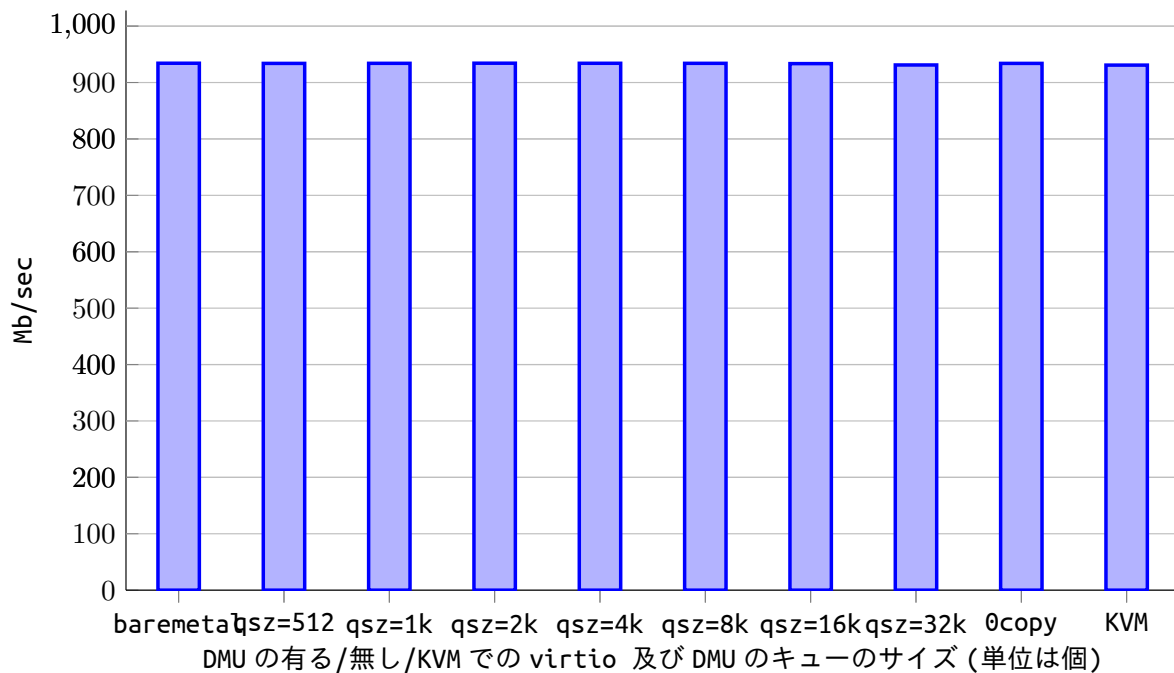


図 4.1 DMU がある場合とない場合、及びある場合のキューサイズごとのスループットの測定

このように、キューのサイズでわずかに性能改善し、最大はキューサイズが 1024 個だった場合だが、いずれの場合にしてもほとんど性能劣化はなく、むしろおそらく偶発的に baremetal の性能を上回った。

これはボトルネックがソフトウェアではなく、ネットワークデバイスそのものにあるということを示している。

4.2 Intel 1GbE でのレイテンシ

次はレイテンシについてベンチマークした。このベンチマークは ping コマンドを flood モードで使うことで行った。ベンチマーク環境については、Appendix A.1 章に記した。

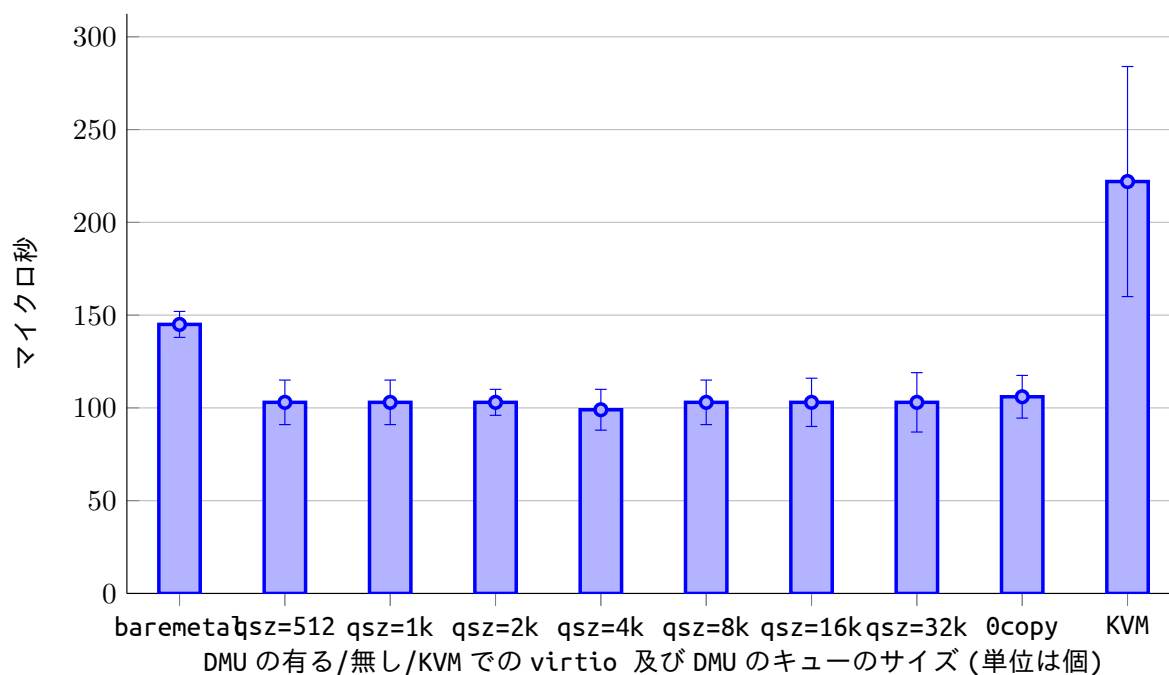


図 4.2 DMU がある場合とない場合と KVM 上のゲストとの通信の場合、及びある場合のキューサイズごとのレイテンシの測定

この図から、やはり KVM の上で動作するオペレーティングシステムとの通信はレイテンシが大きくなってしまふことが分かる。しかし意外なことに、KVM に対してだけではなく、baremetal に対しても Device Masquerading Underlay を使った場合の方がより小さなレイテンシで動作することが分かった。

この事象に関する仮説として、おそらく Linux の割り込みハンドリングの仕組みによるものだということが出来る。

Linux では、割り込みがくるとその原因を調べ、原因をクリアしてから適切なハンドラを選択し登録する。その後、そのハンドラを適切なタイミングで実行する (top/bottom half)。そこで考えられる Linux の 1 GbE ドライバの受信時の推測される挙動を示す。スループットのベンチマークで判明したとおり、ネットワークデバイスの IO がボトルネックであることから、PRO/1000 のリングバッファからイーサネットフレームを取り出す箇所が特に遅いと仮定する。

さらに、Device Masquerading Underlay を用いた場合の Function Kernel, Device Masquerading Underlay, そしてデバイスの推測される挙動も示す。これも上記の仮定を踏まえて作図した。ただし、virtqueue のバッファは Function Kernel が管理しているメモリ領域であり、Device Masquerading Underlay が virtqueue への書き込みを完了した時点でイーサネットフレームはすでに Function Kernel 内にコピーされているので、Function Kernel 内の Bottom Half 処理はすぐに終了する。

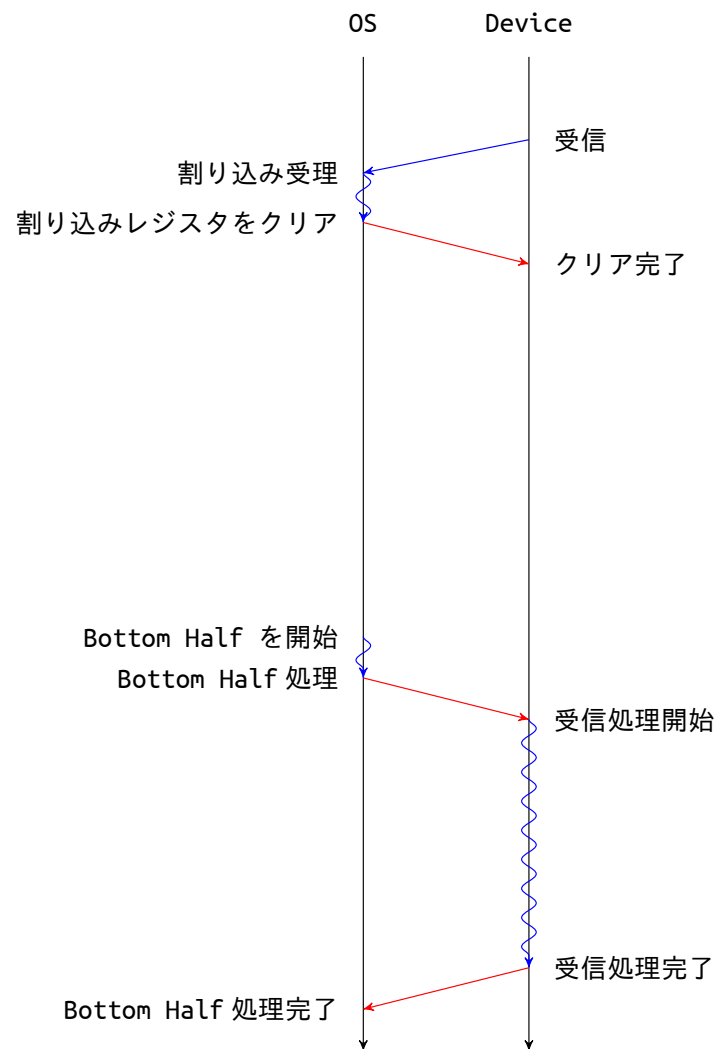


図 4.3 (左) PRO/1000 に対する Linux の挙動 (baremetal)

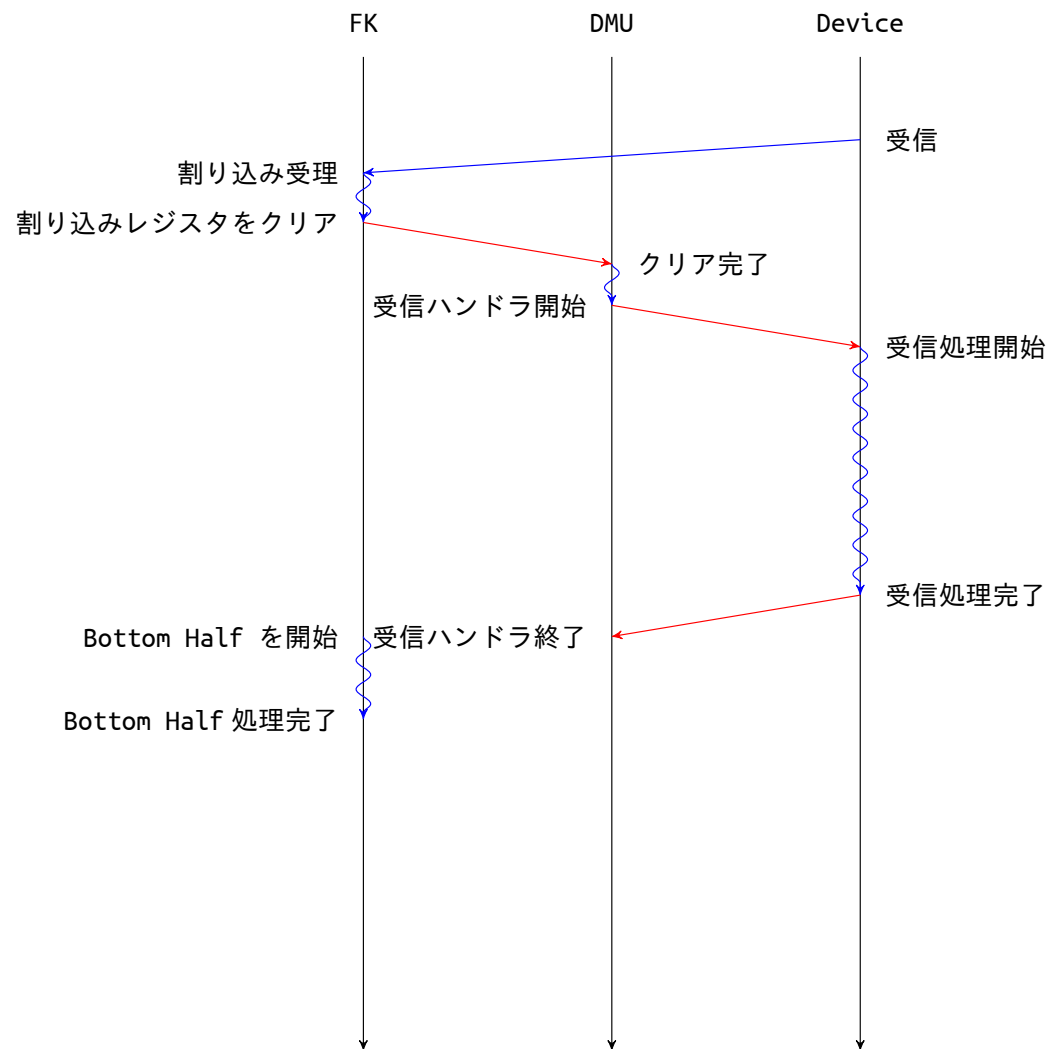


図 4.4 (右) PRO/1000 に対する Device Masquerading Underlay と Function Kernel の挙動

今回の実験結果がこの仮説によるものならば、はからずも `vmx-root` と `vmx-nonroot` の分離を行ない、それぞれを保護しながら並列実行する デバイスドライバが出来ていたことになる。このようにして IO 負荷を Function Kernel から Device Masquerading Underlay へオフロードし、結果として速い応答速度を獲得していたこととなる。

4.3 セキュリティ

本論文での、セキュリティへの貢献は、オペレーティングシステムのカーネル一般のような比較的大きなソフトウェアから attack surface のより小さいソフトウェアにデバイスドライバを移すことで、恒久的な破壊などを主とした攻撃を防ぐというものである。

そこで、仮に LOC 数がセキュリティ上の欠陥に比例関係を持っていると仮定し、Device Masquerading Underlay の LOC 数と一般的なオペレーティングシステムのカーネルの LOC 数とを比較し、これをセキュリティ上の指標としたいと思う。

本論文では、元になったソースコードも含めすべての機能全体で 215 kLOC となっている。それに対し、例えば Linux のソースコードはデバイスドライバを除いて 7.13 MLOC ある。それに対し、Device Masquerading Underlay の実装は、virtio デバイスを作成する箇所やデバッグシェルを除くとほとんどユーザやオペレーティングシステムからの入力を受け付けないので、その分 attack surface が小さいと言える。また、保護機構についても、オペレーティングシステムで使われる Ring 保護だけでなく、EPT/VMCS でのセキュリティ強制機構も同時に働いているため、ハードウェアの不揮発性レジスタなどのアクセスによる破壊を防ぐことが出来る。

前年度の本研究室での先行調査によれば、デバイスへの入力をファジングしている際 [?], ラップトップコンピュータが起動しなくなることがあった。これによって偶発的に見つかった問題は、イーサネットコントローラの不揮発メモリ領域と BIOS の不揮発メモリ領域が参照するフラッシュデバイスが同じものである製品が市場に出回っており、これによってイーサネットコントローラのドライバの不具合や悪意のある侵入者によって、そうしたコンピュータが破壊されてしまうというものだ。

本論文の実装にあたって、ネットワークインターフェースの隠蔽・virtio 化の際に、当該のネットワークインターフェースの PCI configuration 空間の BAR[4] にある不揮発メモリ領域を操作するためのデバイスへのポインタを上書きしているので、この問題は Device Masquerading Underlay を使うと未然に防ぐことが出来る。

結論

5.1 まとめ

本研究では、新しいオペレーティングシステムのデザインを提案し、Proof-of-Concept となる実装を開発した。この新しいオペレーティングシステムアーキテクチャは、デバイスドライバからなる Device Masquerading Underlay と、他の一般的な機能を提供する上位レイヤの Function Kernel からなる二層構造を持っている。Function Kernel 既存のオペレーティングシステムが使える上に、Function Kernel として使われるソフトウェアが変わっても Device Masquerading Underlay は再利用できるようにするため、既に知られているような、デバイスを制御して抽象的なデータに変換するデバイスドライバではなく、個別のデバイスを標準規格で定められた高速で簡潔な仮想デバイスに変換する「デバイス→デバイス変換」を元にした新しいデバイスドライバフレームワークを提案し、この技法を Device Masquerading と名づけた。このアーキテクチャが一般に普及した場合のメリットを以下に挙げる。

- ・ オペレーティングシステムという単一の巨大なシステムソフトウェアから、異なるレイヤの機能を提供する箇所を分離し、関心の分離を図ることができる
- ・ デバイスドライバを分離し、より複雑な機能を持ちうる Function Kernel からデバイスを保護することで、永久破壊に繋がる攻撃パスが可能な attack surface の極小化が可能となる
- ・ デバイスドライバの抽象化方法として新規のメソッドを提案し、これによりデバイスドライバのオペレーティングシステムに対する独立性を大きく引き出す
- ・ デバイスドライバのオペレーティングシステムに対する依存性を下げたことでオペレーティングシステムごとにデバイスドライバを開発する必要がなくなり、デバイスドライバの開発に集中的なリソースを割くことが出来る
- ・ デバイスドライバの開発リソースを纏められるようにしたことで、オペレーティングシステム開発の参入障壁が下がり、新しいオペレーティングシステムが実用に耐えうるものとなった
- ・ オペレーティングシステムによらず単一のデバイスドライバが使えるようになったことで、デバイスドライバの潜在的なバグが発見されやすくなる

また、このレイヤの導入によっても性能が決して大きく劣化することはないばかりか、条件によってはより高速になる事例があることが明らかとなった。

5.2 今後の発展

5.2.1 対応デバイスクラスの追加について

本研究では主にネットワークインターフェースをターゲットに実証試験を行った。これにはいくつかの理由がある。

第一に、ネットワークインターフェースは特に他の普及したインターフェースに比べても統一性のないもので、デバイスドライバの開発コストが特に高いということが挙げられる。第二に、ネットワークインターフェースとして virtio を使うオペレーティングシステムは既に多く知られていて、virtio が定義するいくつかのデバイスクラスの中でも突出して有名である点が挙げられる。また第三に性能評価の指標が明らかで、本実装の優位性を主張しやすいという点があった。

ブロッッククラス

しかし、実際にオペレーティングシステムの動作に不可欠なデバイスクラスはこれだけではない。ネットワーククラスの次に注目されるデバイスクラスとして、ブロッックデバイスが挙げられる。ブロッックデバイスは伝統的には AHCI、SATA や SAS といった統一規格があり、デバイスドライバの開発コストを特に高めている要因ではあまりなかったが、近年のフラッシュメモリを用いた高速ストレージの台頭によって、NVMe、M.2、U.2、Persistent Memory などの規格がこれに加わり、こんにちではすでに決してストレージのデバイスドライバも容易だとは言えなくなっている。virtio にも、ブロッックデバイスに相当するデバイスクラスとして、一般的なブロッックデバイスと、Out-of-Order で通信できる SCSI クラスの二つがある。ストレージに対する Device Masquerading の開発動機は高いといえよう。

グラフィックスについて

また、virtio にはさらに、グラフィックドライバや memory ballooning などが定義されている。グラフィックドライバの開発は商業的な環境によりドキュメントが少なく困難を極めるため Device Masquerading を行なうことも困難と言えるが、こちらは QXL と既存のオペレーティングシステムとの何らかのブリッジを作るなどの手法を考案すると可能になるかもしれない。このアプローチは Le Vasseur らの先行研究 [4] に近い。

Memory ballooning クラス

Memory ballooning は、起動した仮想マシンに対して後からメモリを追加するための機構だが、仮想マシンに限らずオペレーティングシステム一般に対しても、メモリホットスワッピングなどと組み合わせて何らかの意味を与えれば便利な機構を提案できる可能性がある。

5.2.2 Function Kernel の組み合わせについて

本研究では、すでに普及した virtio の古い規格ではなくあえて昨年中に規格の策定が進行中であった virtio 1.0 を対象に実装を行った。これは後述する Exitless Device Masquerading という技法を開発するためだが、virtio 1.0 の規格はまだ草稿段階で、最新版は昨年 10 月に発表されているなど、まだ途上の段階の規格といえる。理論上はほとんどどのようなオペレーティングシステムであっても Function Kernel として使えるとしたが、この事情により、まだ対応しているオペレーティングシステムは少ない。

Virtio の古い規格には非常に多くのオペレーティングシステムが対応しているため、多くのオペレーティングシステムが今年中にも新しい規格にも対応することが予想される。例えば、昨年冬に発表された新しいライブラリ OS である IncludeOS[2] の開発サイト [1] でも virtio 1.0 の開発が今なされていると宣伝されている。これからの開発が待たれる。

5.2.3 Exitless Device Masquerading

ドライバがデバイスへ通知した際や割り込み原因を調べた際、Device Masquerading Underlay がフックするということは既に述べた。しかしフックのために `vmx-nonroot` から `vmx-root` へコンテキストスイッチが起きる。将来的にさらに高速なデバイスに対応する際、このコンテキストスイッチが性能に影響を与える可能性がある。そこで Exitless Device Masquerading という技法を提案する。

Exitless Device Masquerading では、`#VE` という例外を使うことがキーアイデアとなる。

EPT でのアドレス変換の結果、適切なページが見つからないと Device Masquerading Underlay にコンテキストスイッチするという仕様を利用して今まではフックしていたが、EPT ではアドレス変換の結果見つかったページへのポインタを持つエントリに `#VE` 例外を発生させることを表すビットが有効になっていると、この例外が発生し、コンテキストスイッチの代わりに Function Kernel への例外が発生する。ここに例外ハンドラを Device Masquerading Underlay があらかじめ用意し、このハンドラで EPTP スイッチを発行することで、アドレス空間のみを Device Masquerading Underlay に移すことができる。これにより、完全なコンテキストスイッチよりも高速に制御を移すことができる。

実験環境について

A.1 Intel PRO/1000 を用いた実験環境について



ソースコード



PCI configuration 空間の例

ここで示す出力は `sudo lspci -xxxxvvvv` というコマンドで得ることが出来る。

特に冒頭の Vendor ID や Device ID、それから capability list の違いが重要だ。

C.1 Intel PRO/1000

```
00:19.0 Ethernet controller: Intel Corporation Ethernet Connection I218-LM (rev 04)
  Subsystem: Matsushita Electric Industrial Co., Ltd. Device 8338
  Control: I/O+ Mem+ BusMaster+ SpecCycle- MemWINV- VGASnoop- ParErr- Stepping- SERR- FastB2B- DisIN
  Status: Cap+ 66MHz- UDF- FastB2B- ParErr- DEVSEL=fast >TAbort- <TAbort- <MAbort- >SERR- <PERR- INT
  Latency: 0
  Interrupt: pin A routed to IRQ 45
  Region 0: Memory at f7f00000 (32-bit, non-prefetchable) [size=128K]
  Region 1: Memory at f7f44000 (32-bit, non-prefetchable) [size=4K]
  Region 2: I/O ports at f080 [size=32]
  Capabilities: [c8] Power Management version 2
    Flags: PMEClk- DSI+ D1- D2- AuxCurrent=0mA PME(D0+,D1-,D2-,D3hot+,D3cold+)
    Status: D0 NoSoftRst- PME-Enable- DSel=0 DScale=1 PME-
  Capabilities: [d0] MSI: Enable+ Count=1/1 Maskable- 64bit+
    Address: 00000000fee0200c Data: 4183
  Capabilities: [e0] PCI Advanced Features
    AFCap: TP+ FLR+
    AFCtrl: FLR-
    AFStatus: TP-
    Kernel driver in use: e1000e
00: 86 80 5a 15 07 04 10 00 04 00 00 02 00 00 00 00
10: 00 00 f0 f7 00 40 f4 f7 81 f0 00 00 00 00 00 00
20: 00 00 00 00 00 00 00 00 00 00 00 00 f7 10 38 83
30: 00 00 00 00 c8 00 00 00 00 00 00 00 05 01 00 00
40: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
60: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
70: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```

90: 00 00 00 00 00 00 00 00 d8 00 00 00 f8 f0 03 31
a0: 00 00 00 00 00 00 00 00 03 10 00 00 00 00 00 00
b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
c0: 00 00 00 00 00 00 00 00 01 d0 22 c8 00 20 00 07
d0: 05 e0 81 00 0c 20 e0 fe 00 00 00 00 83 41 00 00
e0: 13 00 06 03 00 00 00 00 00 00 00 00 00 00 00 00
f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

```

C.2 virtio-net デバイス

00:19.0 Ethernet controller: Red Hat, Inc Device 1041 (rev 04)

Subsystem: Red Hat, Inc Device 0041

Control: I/O+ Mem+ BusMaster+ SpecCycle- MemWINV- VGASnoop- ParErr- Stepping- SERR- FastB2B- DisIN

Status: Cap+ 66MHz- UDF- FastB2B- ParErr- DEVSEL=fast >TAbort- <TAbort- <MAbort- >SERR- <PERR- INT

Latency: 0

Interrupt: pin A routed to IRQ 20

Region 0: Memory at df200000 (32-bit, non-prefetchable) [size=4K]

Region 1: Memory at df201000 (32-bit, non-prefetchable) [size=4K]

Region 2: Memory at df202000 (32-bit, non-prefetchable) [size=4K]

Region 3: Memory at df203000 (32-bit, non-prefetchable) [size=4K]

Region 4: Memory at df204000 (32-bit, non-prefetchable) [size=4K]

Capabilities: [40] Vendor Specific Information: VirtIO: CommonCfg

BAR=0 offset=00000000 size=00000038

Capabilities: [50] Vendor Specific Information: VirtIO: Notify

BAR=1 offset=00000000 size=00000014

Capabilities: [60] Vendor Specific Information: VirtIO: ISR

BAR=2 offset=00000000 size=00000004

Capabilities: [70] Vendor Specific Information: VirtIO: DeviceCfg

BAR=3 offset=00000000 size=00000038

Capabilities: [80] Vendor Specific Information: VirtIO: <unknown>

BAR=4 offset=00000000 size=00000014

Capabilities: [90] MSI: Enable- Count=2/1 Maskable+ 64bit-

Address: 00000000 Data: 0000

Masking: 00000010 Pending: 00000000

Kernel driver in use: virtio-pci

```

00: f4 1a 41 10 07 00 10 00 04 00 00 02 00 00 00 00
10: 00 00 20 df 00 10 20 df 00 20 20 df 00 30 20 df
20: 00 40 20 df 00 00 00 00 00 00 00 00 00 f4 1a 41 00
30: 00 00 00 00 40 00 00 00 00 00 00 00 00 05 01 00 00
40: 09 50 10 01 00 00 00 00 00 00 00 00 00 38 00 00 00
50: 09 60 10 02 01 00 00 00 00 00 00 00 00 14 00 00 00
60: 09 70 10 03 02 00 00 00 00 00 00 00 00 04 00 00 00
70: 09 80 10 04 03 00 00 00 00 00 00 00 00 38 00 00 00
80: 09 90 14 05 04 00 00 00 00 00 00 00 00 14 00 00 00

```

```
90: 05 00 10 01 00 00 00 00 00 00 00 00 10 00 00 00
a0: 00 00 00 00 00 00 00 00 00 03 10 00 00 00 00 00
b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
c0: 00 00 00 00 00 00 00 00 01 d0 22 c8 00 20 00 07
d0: 05 e0 80 00 00 00 00 00 00 00 00 00 00 00 00 00
e0: 13 00 06 03 00 00 00 00 00 00 00 00 00 00 00 00
f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

参考文献

- [1] Bratterud, A. Includeos. <http://www.includeos.org/>.
- [2] Bratterud, A. Includeos: A minimal, resource efficient unikernel for cloud services. IEEE Cloudcomm'15.
- [3] Chou, A., et al. An empirical study of operating systems errors. In Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (New York, NY, USA, 2001), SOSP '01, ACM, pp. 73--88.
- [4] LeVasseur, J., et al. Unmodified device driver reuse and improved system dependability via virtual machines. In Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6 (Berkeley, CA, USA, 2004), OSDI'04, USENIX Association, pp. 2--2.
- [5] Miller, C. Battery firmware hacking inside the innards of a smart battery. Blackhat USA'11.
- [6] OASIS Open. Virtual i/o device (virtio) version 1.0 committee specification draft 05 / public review draft 05. <http://docs.oasis-open.org/virtio/virtio/v1.0/csprd05/virtio-v1.0-csprd05.pdf>, Oct. 2015.
- [7] Palix, N., et al. Faults in linux: Ten years later. In Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (New York, NY, USA, 2011), ASPLOS XVI, ACM, pp. 305--318.
- [8] Russell, R. Virtio: Towards a de-facto standard for virtual i/o devices. SIGOPS Oper. Syst. Rev. 42, 5 (July 2008), 95--103.
- [9] Shinagawa, T., et al. Bitvisor: A thin hypervisor for enforcing i/o device security. In Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (New York, NY, USA, 2009), VEE '09, ACM, pp. 121--130.
- [10] Weinmann, R.-P. The hidden nemesis - backdooring embedded controllers. PacSec'10.