インターフェーストンネリングによるシステムプログラミング学習

小田謙太郎[†] 和泉 信生[††] 下園 幸一[†] 山之上 卓[†]

システムプログラミング学習の手段として，従来カーネルなどの低レベル環境において提供される入出力などのインターフェースを Java 言語環境などの高レベル言語環境にて提供する手法であるインターフェーストンネリングを提案する．インターフェーストンネリングを行うことで，カーネル空間プログラミングの煩雑なデバック作業やセットアップ作業を最小化し，本質的な問題に集中できるようになる．実際に，Java 言語上に Ethernet デバイスとブロックデバイスのインターフェーストンネリングの実装を行った．前者の実装を用いて，Java 言語にて TCP/IP プロトコルスタックを構築する演習を行い，短期間で TCP の実装に成功し，評価を行った．

Learning System Programming through Interface Tunneling

Kentaro Oda[†] Shinobu Izumi[††] Koichi Shimozono[†] and Takashi Yamanoue[†]

In this paper, we propose a new mechanism called *interface tunneling* which provide low-level interfaces at a high-level language environment such as Java for learning system programming. The low-level interface allows direct access to underlying devices like such as Ethernet network interfaces and block devices, traditionally only available in a kernel space. With the introduction of interface tunneling, we can minimize hassle and burden of system programming often incurred by kernel space development process. To evaluate the effect of interface tunneling, we implemented Ethernet network device and block device tunneled interfaces and conducted TCP/IP protocol stack implementation exercise in Java. Students successfully implemented TCP and UDP protocols, and gave us positive comments.

1. はじめに

「昔は，フロッピーディスクを使っていた．ディスクドライブの仕組みが単純で動作がわかりやすかった．」，「(CPU/OS/プログラミング言語/ミドルウエア）が高性能・高機能になりすぎて，自分が書いたコードのアルゴリズムの良し悪しを実感しにくくなった」といった話がよく聞かれる．ハードウエア，システムプログラム（OS，ミドルウエア）の発展に伴い，様々な抽象化レイヤが追加され，本当にハードウエアを意識せずにプログラミングができるようになってきた．システムプログラムの一つの目的である，ハードウエアの抽象化，仮想化が「成功」したといえる．その一方で，高度な抽象化や仮想化が達成されてくると，システムプログラムの内部の高度な仕組みについて意識されることが少なくなり，その存在が透明になっていく．

システムプログラムの発展とその成功に伴い，システムへの理解は重要性を増している中，学習者にとっては，理解が困難になってきているジレンマを抱えている．この現状を打破する新たな学習環境が必要である．

2. 従来のシステムプログラミング学習方法

OS の働きや仕組みについて，大学の講義において実習形式で学習する方法には，以下の方法が考えられる．

・アクティブ型：プログラミングによる方法
　デバイスドライバを書く
　カーネルを書き換えてみる
・パッシブ型：観測による方法
　システムコールのトレースを行う
　パケットキャプチャのキャプチャ
　デバッガにてカーネルの状況を知る

アクティブ型の特徴は，実システムに触れることで，現実システムの複雑さを直接知ることができる点である．例えば，デバイスドライバを記述するには，巨大な Linux カーネルや PC ハードウエアの詳細な知識が必要になる．これは，現実のシステムを知るには非常に良い手段であるが，詳細の理解に要する時間やバッファオー

† 鹿児島大学　学術情報基盤センター
Kagoshima University, Center for Computing and Communications
†† 崇城大学　情報学部　情報学科
Sojo University, Faculty of Computer and Information Sciences

バーフロー等のプログラミングエラーによるデバッグ時間は無視できない．そのためアクティブ型は，限られた時間では選択できないオプションである．一方，パッシブ型の特徴は，プログラミングを行わないため，デバッグ等でまとまった時間を必要としない点にある．しかしながら学習手段は観測のみであるため，アクティブ型に比べれば学習効果は低いと考えられる．

アクティブ型ながら準備時間及び学習時間の短い手段の提供が必要である．

3. 提案：インターフェーストンネリングによるシステムプログラミング

従来のアクティブ型の問題点の一つは，カーネルプログラミングを行うことで，準備やデバッグに時間がかかることであった．そこで，Java などの高レベル言語環境において，カーネル空間において提供されるディスク入出力やパケット入出力などの低レベルインターフェースを提供することで，システムプログラミング学習の障壁を小さくする方法を提案する．具体的に低レベルインターフェースとは，通常カーネルにおいて提供される以下のようなインターフェースを指している．

- Ethernet のフレームの送信受信
- ブロックデバイスの読み書き
- フレームバッファへアクセス
- USB デバイスへのアクセス
- Linux VFS(Virtual File System)レイヤでのファイルシステム操作
- スケジューリング（プロセス，リソース割り当て）
- ブロックデバイスでの先読みアルゴリズム，キャッシングポリシー
- 物理/仮想メモリ空間, I/O 空間へのアクセス

例えば，Ethernet のデバイスドライバはカーネル内に Ethernet フレームの送信受信インターフェースを提供する．このカーネル内に提供されたインターフェースを，Java 仮想マシン空間から利用できるようにすることで，カーネルプログラミングすることなしに，Java 言語での簡素な記述で Ethernet フレームを直接入出力することが可能になる．同様に，ブロックデバイスドライバがカーネル内に対して提供する入出力インターフェースを，Java にて実装することにより，カーネルに対して，実験的な要素を含んだ仮想的なブロックデバイスを容易に実装することができるようになる．これによって，カーネルのブロックデバイスへの入出力の振る舞いを解析することなどが可能になる．

このようにカーネルにおいて提供されるインターフェースを Java 言語や Ruby 言語などの高レベルな言語環境において提供することを，我々は*インターフェーストンネリング*と新しく命名する．インターフェーストンネリングにより，カーネルやミドルウエアが提供する抽象化機構をバイパスし，現実のシステムの生のハードウエアに直接アクセス可能になる．これによって，実システムへの理解を深めることが可能になる．さらにはカーネルに仮想デバイスを提供することで，カーネルのデバイスドライバに対する振る舞いを詳細に観測できるだけなく，例えば，RAID5 などの冗長性を提供するブロックデバイスの実験的な実装などのアクティブな演習が高レベル言語にて簡素に実現可能になる．

3.1 トンネリングデバイスとの違い

トンネリングデバイスは，デバイスとしてのモデル化に意味があるものが多い．例えば，仮想ネットワークインターフェースや遠隔ディスクといった高レベルな実体は，UNIX ではキャラクタデバイスやブロックデバイスといった相対的に低レベルな実体として提供される．このようにトンネリングデバイスでは，トンネルによるデバイス化が既存の環境との整合性を確保する手段として用いられている．

一方，インターフェーストンネリングは，これとは逆であり，カーネルプログラミング環境を高レベル言語環境にて提供し，高レベル環境での計算が実際にカーネルに反映することが本質である．実際，トンネル対象インターフェースとして，入出力インターフェースのみでなくスケジューリングやキャッシングポリシー等の従来トンネリングデバイスがカバーしなかったもの（デバイス化の意味がなかったもの）を含みえる．一方，トンネリングデバイスは，後述するようにインターフェーストンネリングの実現手段に利用することができる．

4. 実装：Ethernet ネットワークインターフェース(TAP for Java)

4.1 設計

実際に Linux カーネル 2.6 系に対して Java へのインターフェーストンネリング実装を行った．実装では，以下の点を念頭においた．

- 可能な限りカーネルに手を加えない：カーネルは常に進化している．特殊なパッチを当てるなどの作業は最低限にする．
- 様々な OS にて利用可能：様々な OS にてインターフェーストンネリングができることが望ましい．
- 可能な限りシンプルに利用可能なこと

4.2 Ethernet ネットワークインターフェース(TAP for Java)の実装

Ethernet ネットワークインターフェースを Java にて提供する TAP for Java は，次の特長を持つ．

・ ***物理反映オブジェクト***：ネットワークインターフェースとそれをモデル化したオブジェクトとの 1:1 対応づけにより，直感的かつ簡単に Ethernet フレームの入出力が可能
・ Java Native Interface(JNI)を利用した C++によるネイティブライブラリと Java のライブラリから構成
・ カーネルに手を加えない．既存のドライバを利用（Universal TUN/TAP device driver[3)], OpenVPN[5)）
・ 幅広い OS に対応可能： 現時点では Linux 2.6 系にて動作可能．FreeBSD, Mac OS X, Windows は，TAP for Java のネイティブライブラリが現在の時点では未対応だが，OpenVPN [5)]に同梱するドライバが各プラットフォームに存在するため対応可能
・

図 1 に TAP for Java の全体構成を示す．TAP for Java は，TapInterface クラスを Java 言語環境に提供する．そこで提供される TapInterface オブジェクトは，Ethernet ネットワークインターフェース(以下 NIC と略する)をモデル化したもので，以下の性質を持つように設計，実装を行っている．

1． オブジェクトの生成：仮想 NIC の生成
2． オブジェクトの消滅：仮想 NIC の破棄
3． オブジェクトの readFrame メソッドの呼び出し：仮想 NIC に届いたフレームの受信．物理 NIC と仮想 NIC をカーネル内でブリッジ接続することで，LAN に接続された外部からのフレームの受信も可能．
4． オブジェクトの writeFrame メソッドの呼び出し：仮想 NIC へのフレームの送信．物理 NIC と仮想 NIC をカーネル内でブリッジ接続することで，LAN に接続された外部へのフレームの送信も可能．

このように，オブジェクトへの操作が物理 NIC へ，物理 NIC への操作がオブジェクトへと反映する関係にあるようにしている[a]．このような物理的な装置へ 1:1 対応づけされたオブジェクトをここでは物理反映オブジェクトと名付ける．このモデル化によって，直感的に物理デバイスを利用できるようにしている．

a)実際には，TapInterface オブジェクトと直接対応づけられるのは仮想 NIC である．しかし，カーネル内にて物理 NIC とブリッジ接続することにより，事実上仮想 NIC を物理 NIC として利用できるためこの解釈が可能である．

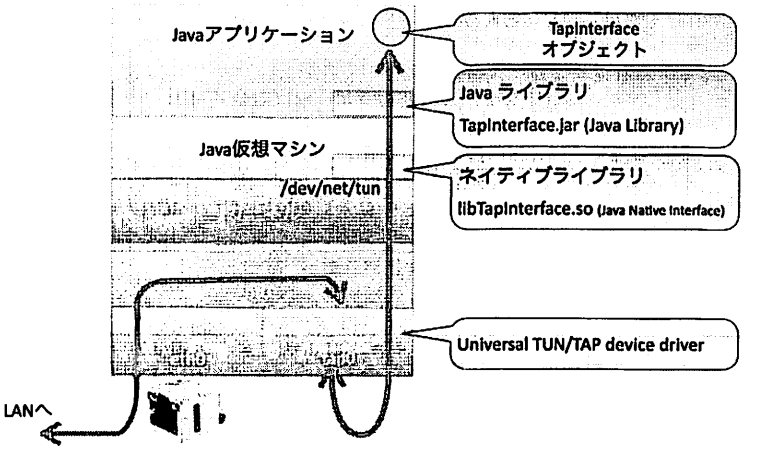

図 1. TAP for Java(黄色の部分)と全体の構成

Ethernet IEEE802.3 フレーム

2	6	6	2	46〜1500	4
プリアンブル	宛先 MACアドレス	送信元 MACアドレス	フレームタイプ	データ/LLC	FCS

読み書き可能

```
byte[] frame = new byte[1514]; // フレームはバイト配列
TapInterface if0 = new TapInterface("tap0"); // NIC生成
int bytesRead = if0.readFrame(frame); // フレーム受信
if0.writeFrame(frame, 0, bytesToWrite); // フレーム送信
```

図 2. TAP for Java の利用方法

図2にTapInterfaceオブジェクトの利用例のコードとEthernet IEEE 802.3フレームにおける読み書き可能な範囲を図示している．このように非常に簡単に任意のEthernetフレームの送受信が可能になる．

4.3 実習例題：リピータの実装

次にTAP for Javaを使った実習の例題としてソフトウエアリピータの実装を挙げる．リピータは，二つのネットワークポートの間でフレームを転送する．学習ブリッジにあるような学習機能は持たない．演習では，二つのNICがあるPCをリピータとして利用しフレームの転送が正しく行われているかを確認する．付録にリピータの完全な実装を挙げる．Repeaterクラスでは，コンストラクタにて，転送を行う二つTapInterfaceオブジェクトの初期化とスレッドの生成を行っている．run()メソッドでは，二つのスレッドがそれぞれの仮想NICからフレームを読み出し，他方のNICへ送り出す処理を行っている．このように非常に少ないコード(全39行)で完全にリピータとして動作する．1000BASE-TのNIC 2つを搭載したIntel Core 2 Duo 2GHz，Sun J2SDK 6の環境にて，80Mbps以上の性能を確認している．

リピータの他に実習例題として考えられるものを挙げる．

- パケットアナライザ：特定のパケット（例えば，ICMPパケットping）をグラフィカルに表示
- ネットワークシミュレータ　：パケット破棄，遅延，順序入れ替え等を設定どおりに行う．
- VLAN，L2学習ブリッジ，　L2 VPN，L3ルータ：Ethernetフレームを直接送受信できるため，VLANタグ付きフレームも送受信可能である．そのためVLANに関連した演習も可能である．高度なスイッチ機能を実現する演習も可能である．特にL2 VPNは，Java環境にて容易にソケットを利用可能なため，簡単に実現可能である．
- プロトコルスタック：TCP/IP(ARP, IP, DHCP, UDP, TCP)などのプロトコルスタックを実装する演習もカーネルプログラミングすることなしに比較的簡単に実現できる．

4.4 TAP for Javaによる利点

インターフェーストンネリングを実現するTAP for Javaにより，以下の利点が得られる．

1. リピータといった従来は簡単に実装できなかった低レベルな機能を簡単に作り込める
2. 高レベル言語によりコンパクトかつ直感的な記述が可能
3. 本質的ではないことに時間をかけることなく，ネットワークプロトコルの実装演習が可能

4.5 TAP for Javaの実際の演習を通じての評価

次にTAP for Javaを利用し実践的な演習を行った．演習は，TCP/IPプロトコルスタックをJavaにて実装するものである．表 に演習の条件と内容を挙げる．演習の途中では，ARP解決やUDPのチェックサム処理，IPパケットのフラグメンテーションといった仕様につまずきが多かった．TCPでは，状態遷移などの振る舞いを理解してからではないと，なかなか実装に入れないため，プロトコルアナライザであるWiresharkなどを用いて学習をおこなった．その実装演習の結果を表2に示す．参加者5名全員がUDPパケットの送受信までは実装できた．参加者全員から演習内容についてポジティブな意見が得られた．

表 1. JavaによるTCP/IPプロトコルスタックの実装演習

学習者	情報工学を学んだ研究室に配属されたばかりの学部4年生3名,博士前期課程1年生3名
学習期間	2ヶ月間にたり 週1回ミーティングと実習
予備演習	IPフレームの表示，リピータの作成，パケット破棄リピータ作成とTCPパケット回復の確認実験(合計3回)
演習	TCP/IPの学習とTCP/IPプロトコルスタックの実装

表 2. 実装演習の結果

学習結果	ARPの実装に成功 5名/5名 UDPの実装に成功 5名/5名 TCPの実装に成功 2名(M1: 2名)/5名
学習者の声	・質問：「この課題によりどのような効果がありましたか.」 1. TCP/IPの仕組みがよく理解できた．(5名) 2. プロトコルスタックの開発について，難しさが理解できた．(5名) 3. 設計に様々な選択肢があることが理解できた．(3名)

	・質問：「この課題のどこが難しかったでしょうか.」 1. プロトコルスタックをどのようにクラスに分割するのかが難しかった. (5名) 2. 効率のよいプロトコルスタックの開発は難しいと思った. (3名) 3. プロトコルの各層をオブジェクトにする設計は遅かった. (1名)

5. ブロックデバイスインターフェース(BlockDevice for Java)

次にブロックデバイスのインターフェースを Java にて提供する BlockDevice for Java(以下 BD for Java)を実装した. BD for Java は次の特長を持つ.

- 仮想ブロックデバイス：通常は，デバイスドライバにより提供される入出力機能を Java オブジェクトにより提供することを可能とした. 具体的には，カーネルの入出力要求を，特定のインターフェース(`BlockDevice`)を実装する Java オブジェクトによって処理することを可能とした.
- 物理反映オブジェクト：ブロックデバイスをオブジェクトとして 1:1 の対応づけによりモデル化し，Linux, Windows, Mac OS X 環境にて生ブロックデバイス操作(Raw block I/O)を提供した. 生ブロックデバイス操作は，カーネルによるキャッシュが無効化された I/O 操作である. 標準の Java API では，生ブロックデバイス操作は提供されないため，ディスクの本来の I/O 性能を知ることは出来ない.
- Java Native Interface(JNI)を利用した C++によるネイティブライブラリと Java のライブラリから構成
- 仮想ブロックデバイスについては，既存のドライバを利用（Network Block Device driver[4]）
- 幅広い OS に対応可能： Linux 2.6 系にて動作. Mac OS 10.5, Windows Xp においては，Raw block I/O を動作確認. Network Block Device driver[4]が存在しない FreeBSD 7 以降[7], Windows 2000SP4 以降[4]は,代わりに iSCSI イニシエータドライバが利用可能であるため，ネイティブライブラリと Java ライブラリの改良により対応可能である.

BD for Java の実装の詳細については，本稿では省略する.

5.1 BD for Java による演習例題

BD for Java を用いて以下のような演習例題が考えられる.

- 仮想ブロックデバイス：ファイルによって仮想ブロックデバイスを実現することや，ネットワーク越しのブロックデバイスの実装，RAID5 といった冗長性をもったブロックデバイスの実現などの演習
- 物理ブロックデバイス（ハードディスク，SSD，CD-ROM 等）の性能評価：様々なアクセスパターンにて，レイテンシやバンド幅による性能評価を行う
- 可視化：仮想ブロックデバイスの実装を工夫し，頻繁にアクセスされるブロック領域（ホットスポット）を直感的に把握出来るようなグラフィカルな可視化を行う

5.2 BD for Java の利点

インターフェーストンネリングを行う BD for Java により,以下の利点が得られる.

1. カーネルからの I/O 要求を，高レベル言語環境にて処理することが可能
2. RAID5 などの冗長機能や，ネットワーク越しの入出力の実験を簡単に実現することが可能
3. カーネルが出力する I/O 要求のアクセスパターンを解析することによって，既存のファイルシステムや I/O サブシステムの評価が可能
4. 物理ブロックデバイスの性能評価が容易に可能

6. 関連研究

システムプログラムの実践を通じた理解を行うことを念頭においた研究としては，教育用 OS[1,2,8]がある. 教育用の OS は，性能よりもコンパクトで理解のしやすさに重点が置かれており，我々の提案と観点が近い. 教育用 OS は，完全な OS の実装をシンプルに包括的に提供している. 我々の手法では，例えばカーネルのコンテクストスイッチを Java 言語のレベルでは実現はできないという意味で，学習出来る範囲が限定されている. 一方，教育用 OS では，現実に利用されている目の前の OS の振る舞いをそこから知ることはできない. インターフェーストンネリングでは，現実の OS の振る舞いを身近に知ることができ，その機能を容易に拡張することができる利点がある.

7. まとめ

現代の OS やミドルウエアなどのシステムプログラムが提供する「厚い抽象化レイヤ」によるハードウエア資源の仮想化，抽象化の成功を述べた. それと表裏一体であるシステムの透明化によるシステムへの理解が困難になるジレンマについて指

摘した．

システムを理解するには，実際にシステムプログラムを行うことが一番であるが，これが非効率的であることを指摘した．これを打開する方法の一つとして，通常カーネルにて提供されるような低レベルインターフェースを，Java 言語などの高レベル言語環境にて提供する，インターフェーストンネリングを提案した．インターフェーストンネリングにより，高レベル言語環境にて，効率的に簡素にデバイスドライバが記述出来ることが期待できる．

そこで実際に，物理 Ethernet ネットワークインターフェースカード(NIC)を Java 環境にてアクセス可能とするインターフェーストンネリングを実装した（TAP for Java）．また，物理ブロックデバイスを Java 環境から直接アクセス可能にするインターフェーストンネリング，Java 環境にてカーネルに対しブロックデバイスを提供するインターフェーストンネリングを実装した（Block Device for Java）．

これによって，簡単に実装できなかった低レベル機能を高レベル言語で実現可能とし，物理デバイスとそれに対応するオブジェクトを 1:1 に対応付けるモデル化によって，システムプログラミングをコンパクトかつ直感的な記述で行えるようにした．

本実装によって，本質的ではないことに時間をかけることなく，多彩な演習が可能であることを示した．実際 TCP/IP プロトコルスタックの実装演習を行い，学習者に好評であった．

今後は，大きな規模で演習を行い学習効果の評価をすることや，現状の実装を改良すること，さらには，割り込みや物理メモリへのアクセス等を含む包括的なカーネルサービスの提供が高レベル言語環境で提供できるかどうかの検討行うことが考えられる．

謝辞　有益なコメントを多く頂いた査読者の皆様，演習実験にご協力頂いた九州工業大学，鹿児島大学の学生の皆様に謹んで感謝の意を表する．

参考文献

1) Andrew S. Tanenbaum, Albert S. Woodhull, OPERATING SYSTEMS DESIGN AND IMPLEMENTATION, Prentice Hall
2) Benjamin Atkin, Emin Gun Sirer, PortOS: An Educational Operating System for the Post-PC Environment, 33rd ACM Technical Symposium on Computer Science Education (SIGCSE), 2002
3) Maxim Krasnyansky, Universal TUN/TAP device driver, http://vtun.sourceforge.net/
4) Microsoft Corporation, iSCSI Software Initiator, http://www.microsoft.com/
5) OpenVPN technologies, Installation Notes SUPPORTED PLATFORMS, http://openvpn.net/index.php/open-source/documentation/install.html
6) Pavel Machek, Steven Whitehouse, Louis D. Langholtz, Paul Clements, Linux kernel 2.6.33.3 nbd.h および nbd.c,
7) The FreeBSD Project, FreeBSD 7.0-REALSE Release Notes iscsi_initiator driver, http://www.freebsd.org/releases/7.0R/relnotes.html
8) 権藤 克彦，大場 勝，中レベル抽象・薄い中間層・追跡性の実践によるコンパクトな教育用オペレーティングシステム udos の設計と実装，電子情報通信学会論文誌，情報・システム J90-D(5), pp. 1194-1208, (2007)

付録

```
class Repeater implements Runnable {
    TapInterface fTap0, fTap1;
    Thread fThread0, fThread1;

    public static void main(String args[]) throws TapException {
        new Repeater();
    }

    public Repeater() throws TapException {
        fTap0 = new TapInterface("tap0");
        fTap1 = new TapInterface("tap1");
        fThread0 = new Thread(this, "Tap0->Tap1 thread");
        fThread1 = new Thread(this, "Tap0<-Tap1 thread");
        fThread0.start();
        fThread1.start();
    }

    public void run() {
        try {
            if (Thread.currentThread() == fThread0) {
                byte[] buff = new byte[2000];
                while (true) {
                    int bytesReceived =
                        fTap0.readFrame(buff, 0, buff.length);
                    fTap1.writeFrame(buff, 0, bytesReceived);
                }
            }
            if (Thread.currentThread() == fThread1) {
                byte[] buff = new byte[2000];
                while (true) {
                    int bytesReceived =
                        fTap1.readFrame(buff, 0, buff.length);
                    fTap0.writeFrame(buff, 0, bytesReceived);
                }
            }
        } catch(TapException te) {
        }
    }
}
```

図 3. TAP for Java によるリピータの完全な実装