社団法人 情報処理学会 研究報告
IPSJ SIG Technical Reports

2007－ＳＬＤＭ－129／2007－ＥＭＢ－4（３）
2007／3／15

通信デバイスドライバの動的な更新手法の提案

田中 裕之† 乃村 能成† 谷口 秀夫†

† 岡山大学大学院 自然科学研究科
(岡山市津島中 3-1-1)
E-mail: †tanaka@swlab.it.okayama-u.ac.jp, ††{nom,tani}@cs.okayama-u.ac.jp

あらまし サービス提供中のシステムを終了させずに，そのシステムの一部を変更できれば，24 時間無停止でのサービス提供が可能となる．既に，実行中プログラムの一部分を入れ替える制御方法が提案されている．しかしこの制御方法はユーザプロセスを対象にしたものであり，オペレーティングシステムのプログラム変更にはそのまま適用できない．そこで，オペレーティングシステムの機能の 1 つである通信デバイスドライバにおいて，旧ドライバから新ドライバに制御を切り替える際に，旧ドライバの動作状態も新ドライバに引き継がせることによって，他に影響をあたえることなく動的に更新する手法を提案する．具体的には，提案方式の実現可能条件を明確にし，FreeBSD の通信デバイスドライバを例に提案手法の実現方式について述べる．

キーワード オペレーティングシステム，ネットワーク，デバイスドライバ，信頼性

A method for run-time updating of network device drivers

Hiroyuki TANAKA†, Yoshinari NOMURA†, and Hideo TANIGUCHI†

† Graduate School of Natural Science and Technology, Okayama University
3-1-1 Tsushima-naka, Okayama
E-mail: †tanaka@swlab.it.okayama-u.ac.jp, ††{nom,tani}@cs.okayama-u.ac.jp

Abstract Run-time updating of software component is a key technology for non-stop systems. We have already proposed a method for updating running user processes, however that method could not be applied to operating systems. In this paper, we propose a new method for run-time updating of network device driver, which provides context migration from an old driver to updated one. This paper also describes requirements for our method and a basic design built on the FreeBSD operating system.

Key words Operating System, network, device driver, non-stop system

1. まえがき

計算機の高性能化と低価格化により，様々なサービスの提供に計算機が導入されていることにともなって，計算機の安定した運用に対する重要性は一段と高まっている．近年では，常時接続ネットワークの普及により，大規模な基幹システムだけではなく，小規模オフィスや家庭向けのサービスにも連続稼働が望まれるものが出現している．このため，今後，計算機サービスの 24 時間無停止運用がますます必要になると考えられる．

一方，計算機サービスの複雑化にともなって，計算機サービスの大半はソフトウェアによって実装されるようになった．従って，サービスの機能向上や問題点修正には，ソフトウェアの修正が必須である．しかしながら，多くの場合，ソフトウェアの変更作業を実行するためには，サービス提供中の計算機を停止させなければならない．すなわち，24 時間無停止でのサービス提供を実現するためには，動作中のソフトウェアを変更する手段が必要となる．

我々は，プロセスとして走行しているプログラムの一部分を変更する方法として，次の大きな 2 つの特徴を持つ方式を提案した[1]．1 つは，サービスプログラムが入れ替えの契機を意識する必要がない点である．もう 1 つは，入れ替え対象のプログラム部分が入れ替え対象ではない別のプログラム部分を呼び出している場合も考慮している点である．更に，複数のプロセス間で共有されたプログラム部分の入れ替え方法を示した[2]．なお，ここでのプログラム部分とは，1 つのプログラムを入れ替えの単位ごとに分割した断片であり，最小単位を関数としている．

これら提案方式を適用することにより，サービスを構成するプログラムのうちユーザプロセスについては動作中の変更が可能となる．しかしながら，これらの手法は，オペレーティング

システム (以降，OS と略す) の提供するプロセス管理やメモリ管理の機能を利用して入れ替え処理を実現しているため，OS そのもののプログラム変更にはそのまま適用できない．

従来，汎用性の高い OS のプログラムでは，動作実績のあるプログラムの再利用が可能なため，サービス開始時点においても高い完成度を期待することができた．また，サービス改変にともなう修正が OS の汎用処理ルーチンにまで波及することは稀なため，OS そのもののプログラム変更を考慮する機会は少なかった．

しかし，近年の計算機ハードウェアの複雑化，および各種入出力デバイスやネットワークシステムの進歩は，これら入出力インタフェースの制御を行なう OS 内のプログラム，すなわちデバイスドライバに対しても，追加改造やバグ修正のための更新を必要とする機会を増大させている．

そこで，本稿では，OS が提供する各種基本サービスのうち，各種ハードウェアとの入出力インタフェースを提供するデバイスドライバに着目し，動作中にそのプログラムを入れ替える方法について提案する．

2. デバイスドライバの更新手法

デバイスドライバの更新方法の種類とその比較を表 1 に示す．更新方法の 1 つは，デバイスドライバの処理を一旦終了させ，プログラムを入れ替えた後再起動する方法である．この方式を終了型と呼ぶ．終了型には，単純に OS のプログラム全体を交換する方式と，交換対象となるデバイスドライバ部分のプログラムのみを交換するモジュール変更方式がある．モジュール変更方式は，近年の OS が備える機能ブロック (モジュール) の動的な挿抜機能 (ローダブルモジュール機能) を利用する方式で，デバイスドライバをローダブルモジュールとして実装することにより実現可能である [3] [4]．

もう 1 つの変更方法は，デバイスドライバの処理を終了させることなくそのプログラムを変更もしくは入れ替える方法である．この方法を非終了型と呼ぶ．プログラム動作中に，メモリ上のプログラムを書き替える方式は，一般にメモリパッチと呼ばれる．ユーザプロセスの交換では，スワップアウトされた外部記憶上のプログラムを書き替える方式も考えられるが，デバイスドライバのプログラムは，通常 OS の一部として動作し外部記憶に退避されることがないため，本稿では検討対象外とする．

表 1 の通り，終了型による更新では，処理が簡単であり，更新の柔軟性も大きいものの，更新時間が長くサービスへの影響も大きいことから，24 時間無停止の運用が必要となるシステムでは採用できない方式といえる．

これに対し，メモリパッチによる更新は，更新時間が短くサービスへの影響も少ないという利点がある．しかしその一方で，動作中の OS がメモリパッチの適用タイミングと適用場所を意識して動作できるようにパッチを適用しなければならないため，処理が複雑で変更の柔軟性が低いという問題がある．

以上に述べた方法の問題を解決するため，デバイスドライバについて，非終了型のモジュール変更方法を考える．本方式は，モジュール全体を更新することから，更新データの作成は容易であり，柔軟性も高い．また，更新時間は，必要最小限のメモリを変更するメモリパッチ方式よりも長くなるが，モジュール本体，もしくは OS 全体の処理を一旦停止させる終了型よりは短くなる．しかし，終了型の更新と異なり，動作中のモジュールの状態を把握し管理した上で，入れ替えの処理を実行しなければならないという問題がある．

文献 [1], [2] では，ユーザプロセスプログラムの走行中における部分的な入れ替えに際しての課題とその対処法を示している．本稿ではこれら文献の内容を踏まえて，動作中のデバイスドライバの動的な更新を実現する，非終了型のデバイスドライバ入れ替え実現に際しての課題とその対処法を提案する．

3. デバイスドライバの動的更新における課題

3.1 ユーザプロセスの入れ替え可能条件

文献 [1] では，図 1 に示すように，入れ替えられるプログラム部分の状態を以下の 3 つに分類している．

未使用 (unused) 実行する前，または実行を終えた状態
走行中 (running) 実行中の状態
呼出中 (calling) 別のプログラム部分を呼び出している状態

未使用 (unused) の状態とは，入れ替え対象となるプログラムを実行する前，または実行を終了した状態であり，図 1 では A の部分を実行中の状態が該当する．走行中 (running) の状態とは，入れ替え対象を実行中の状態であり，図 1 では B の部分を実行中の状態が該当する．呼出中 (calling) の状態とは，入れ替え対象のプログラムから他のプログラムを呼び出している状態であり，図 1 では C の部分を実行中の状態が該当する．

また，文献 [1] では，入れ替え対象プログラムへの必須条件として以下の 2 つを示している．

(条件 1) 呼出しと返却のインタフェースの一致
(条件 2) 処理矛盾の回避 (ループカウンタのリセットなど)

また，呼出中の状態における条件として以下の 2 条件を示している．

(条件 3) 戻り先のアドレスを変更しない
(条件 4) スタティックリンクの場合，メモリ上の外部変数の参照アドレスが同一であること．ダイナミックリンクの場合，プログラム入れ替え時に参照アドレスを解決すること

更に，プログラムの処理内容に大きく依存する項目として以下の 2 条件を示し，変更内容に応じてプログラム毎に個別の対処が必要であるとしている．

(条件 5) 外部変数の値が入れ替え前後で同じであることの保証が必要か否か
(条件 6) アドレス渡しによる外部変数や内部変数の参照と更新がどのように行われているか

3.2 通信デバイスドライバの更新可能条件

3.2.1 更新開始条件

通信デバイスドライバは，システムコールを介して一時に複数のユーザプロセスから実行される．つまり，複数のプロセスから呼び出されるリエントラントな共有プログラムの一種であるといえる．

表 1　デバイスドライバの更新方法とその特徴

Table 1　Method and characteristics of device driver updating

更新方法		対象	変更時間	サービスへの影響	処理の複雑さ	更新データ作成の難易度	変更の柔軟性
終了型	OS 変更	全体	極大	大	簡単	容易	大
	モジュール変更	部分	大	大	簡単	容易	大
非終了型	メモリパッチ	部分	極小	小	複雑	困難	小
	モジュール変更	部分	小	小	複雑	容易	大

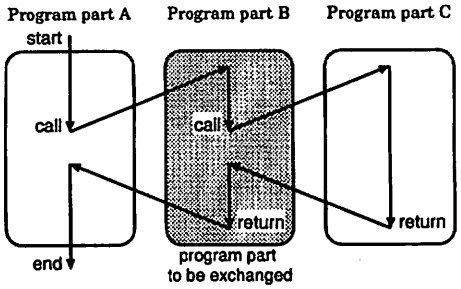

図 1　状態と実行プログラム部分の関係

Fig. 1　Relation between states and program part to be executed

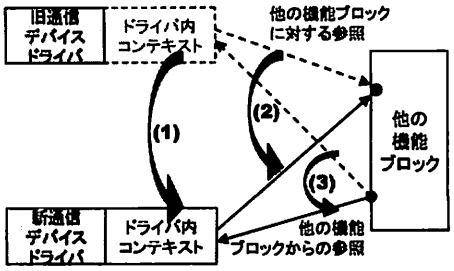

図 2　デバイスドライバのコンテキスト移行

Fig. 2　Context migration with updating device driver

図 1 で入れ替え対象となるプログラム部分 B は，非終了型のデバイスドライバ入れ替えでは，デバイスドライバのプログラムに該当する．同様に，プログラム部分 B を呼びだすプログラム部分 A は入出力システムコールエントリ関数などの OS 内の上位汎用処理モジュール，プログラム部分 B から呼びだされるプログラム部分 C はメモリ割り当て関数やバス制御デバイスなどの OS 内の基本処理プログラムに相当する．

OS では，デバイスドライバの追加実装を容易にするため，デバイスドライバとのインタフェースを明確に規定している．また，モジュールとして正常に動作するためには，処理矛盾を回避する必要がある．すなわち，モジュールとして実装された通信デバイスドライバは，(条件 1)，(条件 2) の両方を満足しているといえる．

ユーザプロセスにおいては，呼出先のプログラム部分 C が処理をブロックして長い時間プログラム B に処理が戻らないという状況が考えられるため，呼出中の入れ替えを考慮する必要がある．一方，通信デバイスドライバが呼び出すプログラムは，長時間のブロックが発生しないメモリの割り当て処理やハードウェアの参照処理である．従って，ここでは，呼出中の状態を入れ替え不可能と規定したうえで (条件 3)，(条件 4) を更新開始条件から除外する．

以上より，通信デバイスドライバが未使用の状態であることが，通信デバイスドライバの更新開始条件となる．

3.2.2　更新時のコンテキスト保持条件

通信デバイスドライバにおける (条件 5)，(条件 6) は，更新対象となるドライバが他の機能ブロックと連携するために参照するドライバ内外の情報，すなわち「コンテキスト」を，各通信デバイスドライバの処理内容および更新内容に応じて，旧ドライバから新ドライバに移行させなければならないことを示している．

通信デバイスドライバのコンテキストは，その参照形態によって以下の 3 種類に区別できる．

（1）　ドライバ内のみで参照されるコンテキスト

（2）　ドライバ外の機能ブロックおよびそのコンテキストに対する参照

（3）　ドライバ外の機能ブロックからの参照

図 2 に，通信デバイスドライバ更新時のコンテキスト移行の概要を示す．これらコンテキストの格納構造の詳細は各通信デバイスドライバの実装によって異なることから，汎用的な手続きで 更新時のコンテキスト引き継ぎを実現することは難しい．しかしながら，新通信デバイスドライバの設計と実装の時点で，更新対象となる旧通信デバイスドライバの実装コードが把握できれば，旧ドライバの所有する各コンテキストについてその格納構造と処理手順を考慮して，システムの動作に支障がでないよう新ドライバに引き継ぐ移行プログラムを実装できる可能性がある．

以上をまとめると，動作中の通信デバイスドライバを更新可能とする条件は，上記 (1),(2),(3) の各コンテンツについて，更新対象となる通信デバイスドライバ以外の機能ブロックからみて破綻がないよう引き継ぎを実行するプログラムが実装可能であることである．

4.　通信デバイスドライバの構造

以下ここでは，FreeBSD の通信デバイスドライバ実装を例に，その動的更新を実現する方法について述べる．FreeBSD

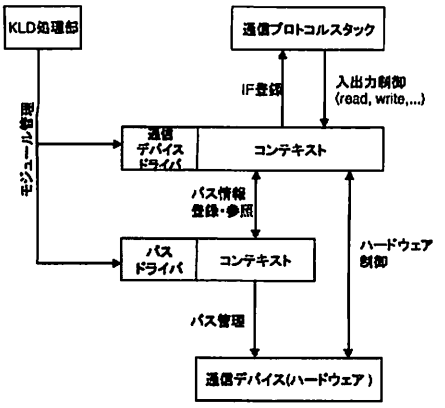

図 3 通信デバイスドライバと関連する機能ブロック
Fig. 3 Communication device driver and related function blocks

は，近年の他 OS と同様，動的な機能モジュールの追加と削除の機能を持つことや，BSD 由来の典型的なネットワークスタック実装を採用していることから，適した事例の 1 つである [5].

4.1 カーネルモジュールによるドライバ実装

FreeBSD では，KLD と呼ばれる，カーネル動作中に動的にモジュールの押抜を実現するインタフェースを用いて，動的に追加と削除が可能なデバイスドライバを実装することができる [3]. KLD では，デバイスドライバの各機能をカーネルから動的に押抜するためのインタフェースとして，以下のような関数を規定している.

probe() ドライバが対応するハードの有無を確認する
attach() ドライバが対応するハードの制御を開始する
detach() ドライバが対応するハードの制御を終了する

多くのデバイスドライバでは，probe() 内で自らが制御可能なデバイスがバス上に存在するか否かを確認するコードを組み込む．また，attach() には，ハードウェアの初期化処理および，カーネル内の他モジュールとのインタフェースの初期設定処理を実装する．逆に，detach() には，attach() で実行した設定の解除および，確保したメモリ領域の解放処理を実装する.

4.2 通信デバイスドライバに関連する機能ブロック

図 3 に通信デバイスドライバと，他の機能ブロックとの相関関係を示す．“通信デバイスドライバ” は，モジュールとしてカーネル内に読み込まれた通信デバイスドライバの実行コードである．また，“通信デバイス (ハードウェア)” は，通信デバイスドライバが制御対象とするハードウェアそのものを示している.

プロセッサと通信デバイスは，PCI などの各種データ伝送路 (バス) を介して相互に接続されている．“バスドライバ” は，これらのバスを制御するデバイスドライバである.

“KLD 処理部” は，カーネルに読み込まれた各モジュールを管理する，カーネル機能部である．“通信プロトコルスタック” は，TCP/IP などを介した通信インタフェースをアプリケーションに提供するカーネル機能部である.

デバイスドライバや，カーネル内の各機能ブロックは，自らの動作状態や，他の機能ブロックの状態を参照するための情報，すなわちコンテキストを動的に保持している．図 3 中の矢印は，デバイスドライバと各機能ブロック間のコンテキスト参照を表している.

KLD 処理部は，前節に示したモジュールの動的な読み込み/解放処理や，それにともなうモジュールの初期化/終了処理の呼び出し処理のため各デバイスドライバのコードおよびコンテキストを参照する.

通信デバイスドライバは，通信プロトコルスタックに自らを通信インタフェースとして登録することによって，カーネルおよびアプリケーションに対して通信デバイス (ハードウェア) による通信機能を提供する．また，通信プロトコルスタックは，通信デバイスドライバを参照することによって通信デバイスを介したデータの送受信を制御する.

バスドライバは，初期化処理の際に，バス上に存在する各デバイスのバス関連情報をコンテキストして収集する．通信デバイスドライバでは，この情報を参照することによって，対象ハードウェアの制御に必要な情報を入手する．また，通信デバイスドライバがハードウェアの制御を開始した時点で，バスドライバは当該ハードウェアの制御ドライバとして通信デバイスドライバを認識する.

5. 通信デバイスドライバの動的な更新手法

5.1 更新処理の手順

終了型のモジュール変更方式，すなわちデバイスドライバモジュールの unload/load による更新処理は，以下のような処理手順となる.

（1） 旧ドライバによる制御を停止
（2） 旧ドライバが確保するメモリ等の資源を解放
（3） 旧ドライバのモジュールをメモリから消去
（4） 新ドライバのモジュールをメモリに読み込む
（5） 新ドライバが必要とするメモリ等の資源を確保
（6） 新ドライバでの制御を開始

3.2.1 節および 3.2.2 節で示した通り，動作中の通信デバイスドライバを他の動作に影響を与えることなく更新するためには，上記処理の途中でコンテキストを引き継ぐ処理を実行しなければならない．また，コンテキストの引き継ぎプログラムを更新前後のドライバの実装にあわせて個々に用意し，引き継ぎ処理の実行前までにメモリに読み込んでおく必要がある．さらに，更新開始条件を満足するため，処理開始時に通信デバイスドライバが未使用の状態を検出しなければならない.

以上の条件を満足するため，非終了型のデバイスドライバ更新では以下のように処理手順を改める.

（1） 新ドライバとコンテキスト引き継ぎプログラムをメモリに読み込む
（2） 旧ドライバが未使用状態になるまで待つ
（3） 旧ドライバの制御を停止 (実行を禁止)
（4） 新ドライバが必要とするメモリ等の資源を確保
（5） 新ドライバに旧ドライバのコンテキストを移行
（6） 旧ドライバが確保するメモリ等の資源を解放

(7) 新ドライバでの制御を開始 (実行を許可)

(8) 旧ドライバのモジュールをメモリから消去

手順 (2) は，通信デバイスドライバが更新可能となる状態を確立するための処理手順である．手順 (5) は，新ドライバによる制御再開時点で，他の動作に影響を与えないようにコンテキストを引き継ぐ．この手順 (5) を実行するためのプログラムを読み込むのが手順 (1) である．

以降では，上記手順に基いた実装を実現する際の課題とその解決策について示す．

5.2 新旧デバイスドライバの共存

KLD では，モジュールのファイル名，またはモジュール作成時に設定するモジュール識別名が同一の場合，カーネル内に複数共存させることができない．そこで，5.1 節で示した新更新手順の実施にあたっては，新モジュールのファイル名とそのモジュール識別名を，旧モジュールとは異なるよう設定することにより，新旧モジュールの共存を実現する．

新モジュールを読み込んだ時点で，ドライバ未割り当てのハードウェアに対する probe が実行されるが，ドライバの更新対象となる通信デバイスについては，旧ドライバが既に割り当てられているため，probe 対象とはならない．従って，新旧ドライバが共存した段階では，ハードウェア制御が競合することはない．

5.3 引き継ぎプログラムの読み込みと実行

コンテキスト引き継ぎプログラムは，引き継ぎ対象となる旧ドライバの実装に合わせて，新ドライバ毎に実装する必要がある．そこで本提案では，新ドライバのモジュールファイル中に新ドライバ用のコンテキスト引き継ぎプログラムを組み込み，モジュールの読み込み処理と同時にメモリに読み込むこととした．

既存システムとの親和性を考えると，旧ドライバが未使用状態になった時点でコンテキスト引き継ぎプログラムを実行する手法についても，新ドライバの実装のみで完結することが望ましい．現在の KLD では，制御ハードウェアが割り当てられないデバイスドライバモジュール内のプログラムを実行可能なタイミングとして，当該モジュール読み込み時の初期登録処理と，新規デバイスドライバ読み込み時点で，ドライバ未割り当てのハードウェアに対して試行される probe 処理の 2 つが存在する．しかし，probe 処理については，既読み込み済の他モジュールのハードウェア制御状況や他既存モジュールの新規読み込みタイミングなどによって，呼び出されるか否かが予測困難なため，コンテキスト引き継ぎ処理を呼び出すトリガとして利用することが難しい．

一方，モジュール読み込み時の初期登録処理については，モジュールのメモリ読み込み時に一度呼び出されるのみであることから，probe 処理よりは容易にコンテキスト引き継ぎのための処理を追加することが可能である．しかし，現在の KLD は，コンテキストの一部であるデバイスドライバ間の相互参照構造を，デバイスドライバ本体のコードから隠蔽していることから，後述するコンテキスト引き継ぎ処理や，旧ドライバを未使用状態に置くための排他制御の実装において，追加制約条件が生じるという問題がある．

以上の問題を踏まえて，本実装では，コンテキスト引き継ぎ処理を実行するインタフェース関数 (以下"handover") をデバイスドライバのインタフェースとして追加するとともに，handover を呼び出して新ドライバに制御を移行するシステムコール (以下 "driver_update") を新規に追加する．

driver_update では，旧ドライバの呼び出しをロックして未使用状態を確保して (処理手順 2,3) 新ドライバの handover インタフェースを呼び出す．新ドライバでは，handover 処理として，新ドライバの初期化 (処理手順 4) と旧ドライバからのコンテキスト移行を実行して新ドライバでの制御開始準備を整える (処理手順 5)．その後，driver_update で，新ドライバの呼び出し処理を許可して (処理手順 6)，旧ドライバの消去を実行する (処理手順 7,8)．

5.4 コンテキストの引き継ぎ

3.2.2 節で示した通り，通信デバイスドライバの動的な更新を実現するためには，ドライバに関連する各コンテキスト (図 3 参照) を，他の機能ブロックからみて破綻がないように，新ドライバの handover インタフェースとして実装しなければならない．

5.4.1 ドライバ内に閉じたコンテキスト

各コンテキストのうち，通信デバイスドライバ内に相関関係が閉じているコンテキストについては，他への影響を考慮しなくてもよいため，移行プログラムの作成が容易である．例えば，旧ドライバの持つ通信ハードの状態管理と設定情報を破棄し，新ドライバにてハードウェアの再初期化を行っても，他の機能ブロックとの相互参照情報が矛盾しない限り問題ない．同様に，KLD 処理のモジュール管理についても，既存の実装を逸脱したモジュールの追加と削除の制御がないことから handover の実装では考慮不要である．

5.4.2 通信プロトコルスタックとの相互参照

通信プロトコルスタックには，各種通信デバイスドライバを通信インタフェース情報 (以降，IF 情報と略す) として登録し，デバイスドライバと通信プロトコルスタックを接続するための登録処理関数が用意されている [6]．通信デバイスドライバがこの登録処理関数を呼び出すと，通信プロトコルスタックは，IF 情報を新規に作成して呼び出し元の通信デバイスドライバの参照情報をその中に保存する．また通信デバイスドライバは，登録処理関数の戻り値として IF 情報への参照情報を入手する．この参照情報はインタフェースの特性設定など，通信デバイスドライバから IF 情報を操作するために，ドライバ側でも保持される．

通信デバイスドライバの更新時のコンテキスト移行対象となるのは，この IF 情報と IF 情報に関連する相互参照となる (図 3 参照)．

IF 情報は，通信プロトコルスタックが主体的に管理する情報のため，通信デバイスドライバからは直接把握できない．プロトコルスタック内部の情報を所持している．これらの内部情報の移行を簡便に実現するため，本実装では，ダミーの IF 情報を使用したコンテキスト引き継ぎ処理を行う．図 4〜6 に，本

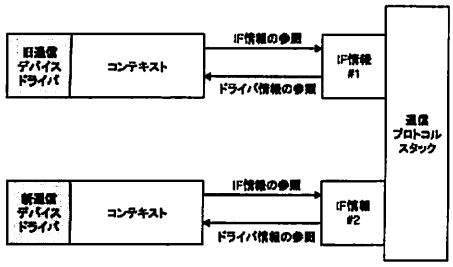

図 4 移行処理 1：新ドライバでダミーの IF 情報を生成
Fig. 4 Phase1: New driver makes dummy IF information.

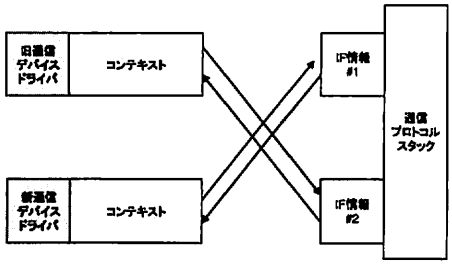

図 5 移行処理 2：新旧ドライバ間で参照を入れ替える
Fig. 5 Phase2: Swap IF references between new and old driver.

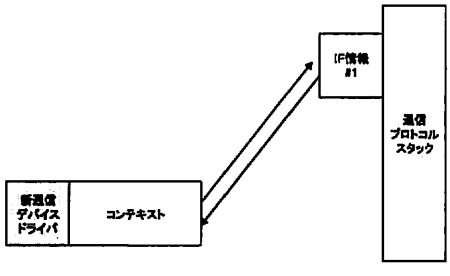

図 6 移行完了：旧ドライバとダミーの IF 情報を削除
Fig. 6 Final: Delete old driver and dummy IF information.

実装による 1 つの IF 情報の引き継ぎ処理の手順を示す．

本実装では，まず，新ドライバからダミーの IF 情報 (IF 情報 #2) を生成する (図 4)．次に旧ドライバが持つ IF 情報#1 と新ドライバで生成した IF 情報#2 の間で参照情報を交換する (図 5)．このとき，旧ドライバ側から登録した参照情報以外の情報も，新ドライバでの必要に応じて合わせて IF 情報#2 に引き継ぐ．この時点で，通信プロトコルスタックに関連するコンテキストの引き継ぎ処理自体は完了する．引き継ぎ処理後に不要となったダミーの IF 情報は，旧ドライバと関連づけられているので，旧デバイスドライバを削除する際に削除される (図 6)．

実際の通信デバイスドライバでは，1 つのデバイスドライバ実行コードが複数の通信インタフェースの制御を行うため，複数の IF 情報と相関を持つ場合がある．複数の IF 情報を引き継ぐ場合は，本手順を IF 情報の数だけ繰り返せばよい．

5.4.3 バスドライバとの相互参照

通信デバイスドライバと，バスドライバとの間で保持しているコンテキストは，バス上のどのハードウェアが，どのデバイスドライバによって制御されているかという，相関関係を保持するための相互参照情報である (図 3 参照)．これらの情報は，通信プロトコルスタックにおける IF 情報と同様，更新対象となる通信デバイスドライバが主体的に管理する情報ではないためドライバ更新時に相互参照情報を引き継ぐことは難しい．

しかしながら，バスドライバとの相互参照情報は，通信デバイスドライバがハードウェアを制御するための情報であり，通信プロトコルスタックや，通信プロトルスタックを介して通信中のアプリケーションの処理に直接影響しない．そこで，本検討では，第一段階として，バスドライバとの相互参照情報についてはコンテキスト引き継ぎ対象外とした．この手法では，処理手順 6 で旧ドライバとの古い相互参照情報を開放した後，処理手順 7 で新ドライバに対する相互参照を新規に構築することとなる．

本手法では，旧ドライバの資源開放をしないと，新ドライバの制御が開始できないため，処理速度の面で改善の余地があるといえる．最終的には，通信プロトコルスタックのコンテキスト引き継ぎと同様，再初期化が不要な手順に変更することが望ましい．

6. まとめと今後の課題

ここでは，OS やアプリケーションの動作に影響を与えることなく動作中の通信デバイスドライバを更新する手法として，新旧の通信デバイスドライバの実行プログラムを一時的に共存させたうえで，旧ドライバから新ドライバに制御を切り替えるタイミングで，旧ドライバの持つ動作状態 (コンテキスト) も新ドライバに引き継がせる手法を提案した．移行対象コンテキストが明確で，かつ旧ドライバが未使用の状態において引き継ぎ処理プログラムが実行可能であることが，提案手法を実現するための条件となる．また，FreeBSD の通信デバイスドライバの実装を例に，提案手法の実現方法について述べた．残された課題として，動的な通信デバイスドライバの更新の実装と評価がある．

文　献

[1] 谷口秀夫，伊藤健一，牛島和夫，伊藤和人，“プロセス走行時におけるプログラムの部分入替え法，” 信学論 (D-I)，vol.J78-D-I，no.5，pp.492-499，May 1995.
[2] 谷口秀夫，後藤真孝，“走行中のプロセス間で共有されたプログラムの部分入替え法，” 信学論 (D-I)，vol.J80-D-I，no.6，pp.495-504，June 1997.
[3] The FreeBSD Documentation Project, “FreeBSD Architecture Handnook,” http://www.freebsd.org/doc/en_US.ISO8859-1/books/arch-handbook/index.html,2000-2006.
[4] Linux Journal Staff, “Kernel Korner: Dynamic Kernels - Modularized Device Drivers,” Linux Journal, Issue 23, no. 7, March 1996.
[5] Marchall Kirk McKusick, George V. Neville-Neil, “The Design and Implementation of the FreeBSD Operation System,” Addition-Wesley, 2005.
[6] G.R. Wright, W.R. Stevens, “TCP/IP Illustrated, Volume 2, The Implementation,” Addison-Wesley, Reading, MA, 1995.