2007－OS－106 (9)
2007／8／3

PBus: 柔軟なカーネル機能拡張のためのインタフェース

平野貴仁[†] 藤田肇[†]
松葉浩也[††] 石川裕[†,††]

PBus は，一般的な Unix 系 OS カーネルに対し，個々の OS に非依存な形で，様々な機能拡張を施すことを可能にする枠組みである．このためには，各 OS が提供する資源の定義を統一しなければならない．本研究報告では，プロセス，スレッド関連の OS 資源に対するインタフェースを設計し，また，そのインタフェースが，カーネルモジュールとしてどのように実装されるかを論ずる．PBus を用いたスケジューラコンポーネントの例を挙げて，PBus 全体の設計，実装の指針とする．

PBus: Interface for Flexible Expansion of Kernel Features

Takahito Hirano,[†] Hajime Fujita,[†] Hiroya Matsuba,[††]
and Yutaka Ishikawa[†,††]

PBus is a framework to expand various features for kernels of common Unix-like OS's, independent of the respective kernels. To achieve this, it is required to standardize the definition of resources provided by them. In this report, we design interfaces for OS resources related to processes and threads, and describe how to implement them. We show an example of a scheduler component which uses PBus, and develop the guidelines for design and implementation of the PBus features.

1. はじめに

低電力高性能 CPU の普及により，従来，PC や高性能計算サーバでしか使われなかった高機能 CPU が，携帯電話，PDA，自動車，ロボット，家電等，いわゆる組込み機器にも使われるようになった．複雑化する組込み機器のライフサイクルを通してのコストを低減するためには，開発コストおよび保守コストの低減が重要である．システムソフトウェアにおけるこれらコストの低減には，新しいデバイスの容易な対応，機能拡張性，瑕疵がないことを検証できるシステムが求められる．我々は，このような機能拡張性，安全性，保守性を提供するオペレーティングシステムの研究開発を行っている．

既存の OS の機能拡張方法には，マイクロカーネル，動的カーネルモジュール，Virtual Machine Monitor (VMM) の 3 つのアプローチがある．マイクロカーネル OS では，CPU の特権モードで動くカーネルは必要最低限の機能を提供し，それ以外の機能はユーザで動くサーバによって提供される[1),2)]．Linux, FreeBSD, Solaris など，Unix 系 OS では，動的カーネルモジュール機能により機能拡張が行える．また，近年，Virtual Machine Monitor (VMM) 上に複数の OS を動かすことにより，レガシ OS の他に機能拡張された OS を同時に実行することも可能になった．これら既存の機能拡張方法では，拡張するための基本機構は提供されているが，OS が扱う資源の機能を容易かつ安全に拡張するためのインターフェイスが提供されているとはいい難い．

そこで，我々は，機能拡張の観点から抽象化された OS 資源をカーネル内で定義する．抽象化された資源上での操作を通して機能拡張モジュールが実装され，動的にカーネルモジュールとして追加可能となる．本インターフェイスを提供するモジュールを PBus と呼んでいる．PBus インターフェイスによる OS 資源の抽象化により，PBus 下に存在するカーネルと機能拡張モジュールは独立する．PBus 下のカーネルの種類やバージョンの違いは PBus によって吸収される．

本論文では，まず，PBus 設計の概略について述べる．次に，OS 資源のうち，プロセスやスレッドといったスケジューラに関わる資源に対する PBus インタフェースの設計と実装について論ずる．PBus インタフェースを用いたスケジューラ例を示すことにより，PBus の全体設計について考察する．

† 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo
†† 東京大学情報基盤センター
Information Technology Center, The Univversity of Tokyo

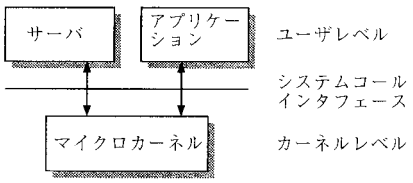

図 1　マイクロカーネルの構造

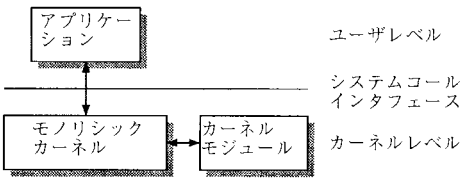

図 2　モノリシックカーネルの構造

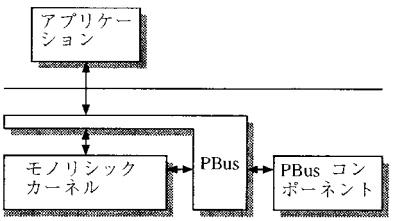

図 3　PBus の構造

2. PBus の概要

OS の機能拡張の手法は, 従来からマイクロカーネルと関連付けて活発に研究されてきた[1),2)]. マイクロカーネル OS では, CPU の特権モードで動くカーネル本体は必要最低限な機能のみに抑え, それ以外の機能は, 非特権モードで動くサーバによって提供を行うという構造になっている. これにより, メモリ保護機構の下で, 安全に機能拡張を行うことができるが, 一方で, モード間の変更や, 通信に伴うコストがかかる (図 1).

Linux, FreeBSD, Solaris などの Unix 系 OS では, 動的カーネルモジュール機構によりカーネル本体の機能を拡張することが可能である. カーネルモジュールは, 一般に特権モードにおける動作となるため, モードの切り替えコストは不要であり, 通信も関数呼び出しなどによるため高速である. しかし, 拡張出来る機能は限られており, カーネルソースプログラムを変更したり, バイナリパッチを当てるという手法が取られている. また, カーネル内の資源を扱うためには, 扱う資源特有のインタフェースを熟知しておく必要があり, いくつかのカーネルに対してカーネルモジュールの提供を行おうとすると, それぞれの OS 固有の設計及び実装が必要となる (図 2).

本論文で提案する PBus インターフェイスにより, Linux をはじめとする Unix 系 OS カーネル固有の構造体や手続きが隠蔽され, オペレーティングシステムを容易, 柔軟, かつ, 安全に変更できる枠組が提供される. PBus は, PBus を用いるカーネルモジュールである「PBus コンポーネント」とカーネル本体を仲介する. PBus 自体カーネルモジュールとして実現される (図 3). 現在, PBus は, Linux カーネル上にプロトタイプを設計実装中であるが, 最終的には, 以下の機能を提供する.

(1) PBus コンポーネントの信頼性保証. PBus コンポーネントを静的に検証する技術が導入できるように, 扱う資源の状態と操作の形式的意味論を与える.

(2) PBus コンポーネントのカーネル組み込み. PBus コンポーネントは動的に組み込まれるだけでなく, PBus 下の OS カーネルソースに組み込むコンパイラを提供する.

(3) PBus コンポーネントのポータビリティ. PBus インターフェイスを様々な Unix 系 OS に移植することにより, PBus コンポーネントが Linux 以外の Unix 系 OS でも利用できるようにする.

3. 設　　計

この節では, プロセス・スレッド関連のインタフェースについて, 資源インタフェースと, それに基づく機能拡張インタフェース, 属性追加インタフェースの 3 つに分けて論ずる.

3.1 プロセス・スレッド関連の資源インタフェース

3.1.1 プロセス

プロセスとは, 一意なプロセス ID を持ち, 独立した仮想メモリ, ユーザ, ディレクトリツリー, ファイルディスクリプタテーブル, シグナルハンドラ, 制御端末を持てる最小単位である. またユーザ空間から `setrlimit` でリソース制限をしたり, `nice` でタイムスライスの調整をしたりすることができる最小単位でもある. プロセスは生成元を親, 生成先を子とする親子関係をなす. 親が終了した場合は, プロセス ID 1 のプロセスの子に再割り当てされる. 各プロセスは, 制御端末の共有を行う範囲であるセッション, 制御端末関連のシグナルが送られる範囲であるプロセスグループに属する. セッション, プロセスグループはそれぞれ一意な ID を持ち, それぞれセッションリーダ, プロセスグループリーダと呼ばれるプロセスを持つ. プロセスグループリーダのプロセス ID は, プロセスグループ ID と等しい.

この定義に従って, PBus が提供を行うインタフェースを **表 1** に示す.

なお, Linux の POSIX スレッドライブラリである NPTL の実装では, バージョン 2.6.20 現在で, ユーザ

表 1　プロセスに関するインタフェース

関数	説明
pbus_proc_start	プロセスの開始
pbus_proc_exit	プロセスの終了
pbus_proc_pid	プロセス ID の取得
pbus_proc_prev	全プロセスリストにおける前のプロセスの取得
pbus_proc_next	全プロセスリストにおける次のプロセスの取得
pbus_proc_parent	親プロセスの取得
pbus_proc_sibling_next	兄弟プロセスリストにおける前プロセスの取得
pbus_proc_sibling_next	兄弟プロセスリストにおける次のプロセスの取得
pbus_proc_thread	プロセスに属する最初のスレッドの取得
pbus_proc_sid	セッション ID の取得
pbus_proc_set_sid	セッション ID の設定
pbus_proc_pgid	属するプロセスグループ ID の取得
pbus_proc_set_pgid	属するプロセスグループ ID の設定
pbus_proc_pg_prev	属するプロセスグループのプロセスリストにおける前のプロセスの取得
pbus_proc_pg_next	属するプロセスグループのプロセスリストにおける次のプロセスの取得
pbus_proc_tty	制御端末の取得
pbus_proc_exitstatus	終了ステータスの取得
pbus_proc_uid	ユーザ ID の取得
pbus_proc_set_uid	ユーザ ID の設定
pbus_proc_euid	ユーザ ID の取得
pbus_proc_set_euid	ユーザ ID の設定
pbus_proc_gid	グループ ID の取得
pbus_proc_set_gid	グループ ID の設定
pbus_proc_egid	グループ ID の取得
pbus_proc_set_egid	グループ ID の設定
pbus_proc_sigaction	シグナルハンドラの設定
pbus_proc_rusage	リソース使用量の取得
pbus_proc_rlimit	リソース制限の取得
pbus_proc_set_rlimit	リソース制限の設定
pbus_proc_vm	仮想メモリ空間情報の取得
pbus_proc_fs	ファイルシステム情報の取得
pbus_proc_fd	ファイルディスクリプタ情報の取得
pbus_proc_nice	nice 値の取得
pbus_proc_set_nice	nice 値の設定

情報, nice 値をスレッドごとに持つことができるなど, この定義に従っていないが, ここでは POSIX 標準に合わせる.

3.1.2　スレッド

スレッドは, 一般に, スケジューラが CPU 時間の配分を行う最小の単位である. スレッドモデルには,

(1)　プロセス 1 個につき, ユーザ空間に N 個, カーネル空間 1 個のスレッドを持ち, ユーザレベルでスケジューリングを行う N:1 モデル

(2)　ユーザ空間のスレッド 1 個が, そのままカーネル空間のスレッド 1 個に対応付けられ, カーネルレベルでスケジューリングを行う 1:1 モデル

(3)　それらを組み合わせた N:M モデル

がある. N:1 モデルではマルチプロセッサ環境を生かせないため, 汎用的な Unix 系 OS では 1:1 モデルや N:M モデルが可能となっている. ここでのスレッドとは, カーネルレベルでのスケジューリングの対象となるスレッドを指す.

スレッドは, スケジューリングの最小単位として, 実行の可否, スケジューリングポリシやプライオリティ, プロセッサアフィニティといった属性に加えて, シグナルの許可, 禁止を行うシグナルマスクを持つ. シグナルの禁止, 許可は, ユーザ空間で行う場合もあるが, その場合は, カーネル空間のシグナルマスクはすべてのシグナルを許可するようになっている.

まとめると, スレッドは **表 2** のようなインタフェースを持つべきことになる.

3.1.3　プロセス ID 空間

プロセス ID 空間も統一的な管理が必要であり, 資源として扱うべきである. PBus は, プロセス ID 空間に対する, **表 3** のインタフェースの提供を行う.

3.1.4　CPU

一つの CPU は, ハイパースレッディングやマルチコアなどによって, カーネルからは二つ以上の CPU に見えることがある. Linux に倣い, 前者の CPU を物理 CPU , 後者の CPU を論理 CPU と呼ぶ. 物理 CPU 群は, メモリアクセスコストの違いからドメインやノードに分けられ, 全体として, 論理 CPU は, ノード, 物理 CPU, コア, 論理 CPU というような階層構造をなす.

また, CPU は起動中に接続, 切断することが可能な場合もある. これは, 大規模なマルチプロセッサシステムで, 障害回避などのために用いられる. これによって, スケジューリングや, カーネルスレッドの配置に影響を与える可能性があるため, コールバック機構が設けられる.

PBus のプロセッサに対するインタフェースは, **表 4** の通りとなる.

3.2　プロセス・スレッド関連の機能拡張インタフェース

Unix 系システムに従来から備わる典型的な機能拡張のためのインタフェースとしては, バーチャルファイルシステムがある. この機構では, バーチャルファイルシステムに対するのオペレーションを各カーネルモジュールで実装されたファイルシステムに移譲することにより, ファイルシステムの追加を実現している.

PBus では, より様々な機能拡張のためのインタフェースの提供を行う. ここでは, スケジューラの置換のためのインタフェースについて述べる.

3.2.1　スケジューラのインタフェース

PBus は, バーチャルファイルシステムと同様に,

表 2　スレッドに関するインタフェース

関数	説明
pbus_thread_current	現在実行中のスレッドの取得
pbus_thread_proc	属するプロセスの取得
pbus_thread_next	属するプロセスのスレッドリストにおける次のスレッドの取得
pbus_thread_prev	属するプロセスのスレッドリストにおける前のスレッドの取得
pbus_thread_runnable	スレッドの実行の可否を取得
pbus_thread_set_runnable	スレッドの実行の可否を設定
pbus_thread_interruptible	スレッドの割り込みの可否を取得
pbus_thread_set_interruptible	スレッドの割り込みの可否を設定
pbus_thread_preemptible	スレッドのプリエンプションの可否を取得
pbus_thread_set_preemptible	スレッドのプリエンプションの可否を設定
pbus_thread_needresched	再スケジューリングの必要の有無の取得
pbus_thread_set_needresched	再スケジューリングの必要の有無の設定
pbus_thread_sigpend	ペンディングシグナルの取得
pbus_thread_set_sigpend	ペンディングシグナルの設定
pbus_thread_sigmask	シグナルマスクの取得
pbus_thread_set_sigmask	シグナルマスクの設定
pbus_thread_policy	スケジューリングポリシの取得
pbus_thread_set_policy	スケジューリングポリシの設定
pbus_thread_affinity	アフィニティの取得
pbus_thread_set_affinity	アフィニティの設定
pbus_thread_priority	プライオリティの取得
pbus_thread_set_priority	プライオリティの設定

表 3　プロセス ID 空間に関するインタフェース

関数	説明
pbus_pid_alloc	PID の取得
pbus_pid_free	PID の解放
pbus_pid_bind	プロセスをプロセス ID に束縛
pbus_pid_find	あるプロセス ID のプロセスの検索

PBus スケジューラに対するオペレーションを各 PBus モジュールで実装された PBus コンポーネントスケジューラに移譲することにより, スケジューラの変更を実現している.

スケジューラは, スケジューリングの機会が与えられるたびに, 次の実行スレッドの選択を行う. 次の実行スレッドの適切な選択を行うためには, 時刻の進行や, 各スレッドの状態変化に対する通知が必要である.

スケジューラで実装可能なコールバックルーチンは,

表 4　CPU に関するインタフェース

関数	説明
pbus_cpu_current	現在実行中の CPU の取得
pbus_cpu_number	稼働中 CPU 数の取得
pbus_cpu_first	最初の稼働中 CPU の取得
pbus_cpu_next	次の稼働中 CPU の取得
pbus_cpu_node	属するノードの取得
pbus_cpu_phys	属する物理 CPU の取得
pbus_cpu_core	属するコアの取得
pbus_cpu_add_notifier	稼働中 CPU 変更の通知関数の追加
pbus_cpu_rem_notifier	稼働中 CPU 変更の通知関数の削除

表 5　スケジューラで実装するコールバックルーチン

関数	説明
clock	タイマ割り込みの通知
choose	次にスケジューリングされるスレッドの選択
add	スケジューリング可能なスレッドの追加
remove	スケジューリング可能なスレッドの削除

表 5 に示した. プライオリティやプロセッサアフィニティ, スケジューリングポリシといった属性が変更される場合は, いったんスレッドのスケジューリングから除外し, スケジューラへの再投入することで, スケジューラが持つスレッドキューの更新を助ける.

3.3　属性追加インタフェース

PBus コンポーネントによって置き換えられたスケジューラが, スレッドに対して独自の統計情報の設定, 取得を行うなど, 既存の資源に対し新たな属性の設定, 取得を行うインタフェースが必要になる場面は多い.

PBus は, PBus コンポーネントにより, 既存の資源に対し, 属性の設定, 取得を行うインタフェースの追加のための支援機構の提供を行う. PBus コンポーネントが動的にロードされるにもかかわらず, 追加されたインタフェースは, 図 4 のように, 既存のインタフェースとほとんど区別をせずに使うことができる. すなわち, 新たに導入された属性は, あたかも最初からシステムによって提供されているかのように見える.

4. PBus の実装

4.1　要　　件

PBus は, それ自体カーネルモジュールの形で作られる. その目的は三つあり, 一つめは, カーネルイメージ本体の変更の回避である. システム管理者によっては, カーネルモジュールによる機能拡張のみ許可という方針をとっていることが少なくない. PBus 自体をカーネルモジュールにすることによって, こういったシステムにも導入を可能にする.

二つめは, 停止が許されないシステムにおいて導入を可能にするためである. 基幹サーバなどの重要なシステムにおいては, 再起動の回避が導入の重要な要件

```
/* 独自属性の宣言 */
PBUS_FIELD(pbus_thread, /* 資源 */
   deadline, /* 属性 */
   unsigned long long);
    /* 属性の型 */

/* 属性の取得 */
pbus_thread_deadline(thread);

/* 属性の設定 */
pbus_thread_set_deadline(thread,
       1000000ULL);
```

図 4　追加インタフェースの使用方法

となる.

三つめは, テストを容易にできるようにすることである. 再起動せずに PBus のインタフェースの評価が可能になる.

しかしながら, スケジューラの変更手段をモジュールで提供するには, いくつかの実装上の問題点がある. ここでは, Linux の場合を例にとって, その問題点と解決法を示す.

4.2 PBus スケジューラ

Linux では, 現在のスレッドの状態の更新, 次にスケジューリングすべきスレッドの決定, そのスレッドへのタスクの切り替えは, `schedule` 関数で実装されている. PBus コンポーネントでスケジューラが実装できるようにするためには, PBus が, この `schedule` 関数を置き換える必要がある.

関数の置き換えについては, 命令の動的書き換え手法, すなわち, 古い関数の先頭に新しい関数へのジャンプ命令を書き込む操作を行う. 本操作を安全に行うには, 次の三つの条件が満たされている必要がある.

(1) 書き換えるコード列付近を実行するスレッドが存在しないこと. これを「書き換え時非実行性」と呼ぶことにする.

(2) 書き換えたコード列に存在していた元のコード列を実行しようとし, 不正なコードが実行されてしまわないこと. 「書き換え後非実行性」と呼ぶことにする.

(3) 新しい関数と古い関数が共存できること. これを「新旧関数共存性」と呼ぶことにする. 以下, 上記 3 つの条件を詳しく述べる.

4.2.1 書き換え時非実行性

書き換え時に, その部分の付近を実行中のスレッドがあってはならない. もしも, 実行中のスレッドがいた場合, 未定義の動作を引き起こす可能性がある. これは, 全スレッドがスケジューリングの実行待ち状態, あるいは休眠状態であることの保証があれば十分である.

シングル CPU の場合, 動的書き換えを行うスレッド以外は, 実行待ち状態, あるいは休眠状態であり, また動的書き換えを行うスレッドは書き換えたい部分を実行中でないことは自明であるので, 常に保証がなされていると考えてよい.

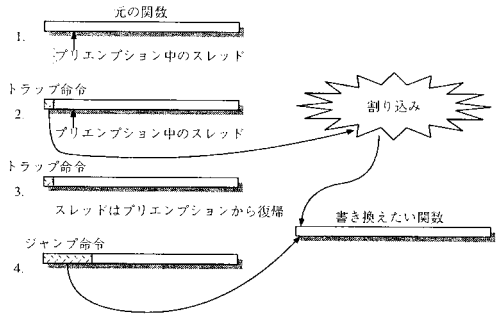

図 5　関数の書き換えの手順

一方, マルチ CPU の場合は, 何もしないスレッドを走らせて, 書き換えを行うスレッドがいる CPU 以外に張り付けておくことで保証が可能となる. Linux では, このために `kstopmachine` という機構がすでにあり, これを用いる.

4.2.2 書き換え後非実行性

書き換え後に, 書き換えた部分の中間に戻ってくるスレッドがあってはならない. `i386` や `amd64` といった, 非固定長命令のアーキテクチャにおいては, 短い命令群の上に長いジャンプ命令を書き込むことになる可能性がある. 関数やループから戻ってくる場合は, アセンブリ段階で静的な保証が可能であるが, プリエンプタブルカーネルにおいては, プリエンプションから戻ってくる場合があり, この場合については静的な保証は不可能である.

そこで, プリエンプタブルカーネルにおいてはジャンプ命令の代わりに一旦トラップ命令を書き込む. ここでトラップ命令とは, 一般的なアーキテクチャにおいてデバッグ目的などのために備わっている例外を引き起こす命令であり, `i386` や `amd64` では, `int3` 命令を指す. この命令は, ジャンプ命令と異なり, どのような命令も置き換え可能な短い命令になっている. このため, その中間にジャンプしてくるといったようなことは起こり得ない. この命令により呼び出された例外ハンドラにおいて, 発生元のプログラムカウンタを古い関数のアドレスから新しい関数のアドレスへと書き換えておくことで, ジャンプ命令と同様の動作を実現している. しかしながら, トラップ命令はジャンプ命令と比べてはるかにコストがかかる. プリエンプションからジャンプ命令が入るべき場所へ戻る可能性がなくなったと保証できた時点で, ジャンプ命令へと置き換えられる (図 5).

4.2.3 新旧関数共存性

新しい関数と古い関数が共存可能でなければならない. これは, 古い関数の途中でタスク切り替えをする

可能性がない場合は考えなくてもよい. 古い関数の途中でタスク切り替えが行われた場合, 切り替えられたスレッドが実行待ち又は休眠状態になる. そのときに, 前の 2 条件を満たし, 動的書き換えが起こる可能性がある. そのスレッドは, 実行待ちから戻ると, 関数を抜けるまでは新しい関数ではなく古い関数の実行を続ける.

schedule 関数は, タスク切り替えを行う可能性がある関数として最たる例である. 関数の書き換えが行われた場合, 古い関数と新しい関数の併存がしばらく続く.

新しい schedule 関数を, 古い schedule 関数と共存できるようにするために, 次のような手法を用いている. 古い schedule 関数は, 新しい schedule 関数の中身を知らないので, 新しい関数で実行待ち状態にあるスレッドの実行の再開はできず, 古い関数で待ち状態にあるスレッドの実行のみスケジューリングするほかない. 一方, 新しい schedule 関数は, 古い schedule 関数の中身を知ることができる. したがって, 新しい関数の中で待ち状態にあるスレッドだけでなく, 古い関数で待ち状態にあるスレッドも, スケジューリングすることが可能である. これにより, 古い関数で待ち状態であるスレッドも, 必要に応じて古い関数から抜け, 徐々に新しい関数が使われるようになっていく.

4.3 属性追加インタフェース

PBus は, 任意の資源に属性を追加できるようにする. 任意の資源で共通に利用できる値として, 資源へのポインタ値がある. これを利用して, 資源へのポインタを鍵にしたハッシュ表を作り, これに追加属性を格納する. 資源へのポインタでハッシュ表を引くことで, 属性の値の取得, 設定を行う.

属性があたかもカーネルに元々あったかのように見せるためには, 属性のメモリ領域の解放が, 資源のメモリ領域の解放と連動して起こらなければならない. このために, メモリ領域の解放を行う関数を, 第 4.2 節で述べたのと同様の手法により書き換えて, 属性の解放も同時に行うようにする.

5. PBus コンポーネントの例

PBus を用いた EDF スケジューラのコードを付録 A.1 に示す. 第 3.3 節で示した追加インタフェース支援機構により, deadline の設定, 取得のインタフェースが加えられている. deadline の 設定を行う pbus_thread_set_deadline 関数を呼び出して, スレッドをスケジューラに投入すると, mysched_add 関数が呼ばれて, プライオリティキューの順番が変更される. 結果として, スレッドは, deadline の小さい方から先にスケジューリングされる. デッドラインの継承などは行っていない.

6. 関連研究

SPIN[3] マイクロカーネルでは, サービスの一部をカーネル空間で動かすことにより高速化を図っている. この機構によりスケジューラも実現可能である.

PlugSched[4] では, Linux のスケジューラを静的に置き換えることができる. PBus では, より一般的に Unix 系 OS のスケジューラを動的に置き換えることを目指している.

OS の動的更新には, 更新可能単位, 安全ポイント, 状態追跡, 状態遷移, 呼び出しのリダイレクト, バージョン管理が必要とされると指摘されている[5]. これらの要求に対して, 現在 PBus は, 以下の特徴を持つ.

- 更新可能単位は, PBus コンポーネントである. PBus コンポーネントは, 既存のカーネルモジュールの仕組みをほぼそのまま用いている. ソースコードレベルでは OS 非依存となるよう工夫がなされている.
- 状態追跡, 状態遷移の実現のために, カーネル本体の関数や変数を読み書きを行う.
- 呼び出しのリダイレクトの実現のためには, カーネル本体の関数書き換えを行う.
- PBus コンポーネントのバージョン管理についてはまだ考えられていないが, 今後開発を活発化させていく中で考える必要が出てくると考えられる.

7. おわりに

プロセスやスレッドといったスケジューラに関わる資源に対するインタフェースの設計や実装により, OS に依存しない様式でスケジューラの機能拡張を行えることを示した. プロセスやスレッドに見られるように, 各 OS における資源の扱いは, 一般的な Unix 系 OS であれば類似性がある. PBus の全体について考えても, インターフェースを適切に定義すれば、 PBus はその差違を吸収することができる. これにより, PBus は, コンポーネントに対し抽象化した OS 像を見せることになる. また, この機能拡張の実現のためには, 広く適用可能な関数の書き換え手法も有用であろう.

PBus は, まだ開発途上である. スケジューラ以外の機能拡張として, メモリ管理, ファイルシステムなどを考えていく. また, インタフェースの設計と実装を進め, さらにその経験からインタフェースの改良などを行っていかなければならない.

謝辞

本研究の一部は, 科学技術振興機構 戦略的創造研究推進事業 (CREST) (領域名：実用化を目指した組込みシステム用ディペンダブル・オペレーティングシステム) 技術課題：「高信頼組込みシングルシステムイメージ OS」 による.

参 考 文 献

1) Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A. and Young, M.: Mach: a new kernel fountation for UNIX Development, *Proceedings of the USENIX Summer Conference*, Altanta, GA, USA, USENIX Association, pp.93–112 (1986).
2) Liedtke, J.: On micro-kernel construction, *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, Copper Mountain, CO, USA, ACM Press, pp. 237–250 (1995).
3) Bershad, B. N., Chambers, C., Eggers, S., Maeda, C., McNamee, D., Pardyak, P., Savage, S. and Sirer, E.G.: SPIN: an extensible microkernel for application-specific operating system services, Technical Report 94-03-03, University of Washington Computer Science and Engineering, Seattle, WA, USA (1994).
4) Williams, P.: CPU Scheduler Evaluation, `http://cpuse.sourceforge.net/`.
5) Baumann, A., Heiser, G., Appavoo, J., Silva, D. D., Krieger, O., Wisniewski, R. W. and Kerr, J.: Providing dynamic update in an operating system, *ATEC'05: Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference*, Anaheim, CA, USA, USENIX Association, pp.271–291 (2005).

付　　録

A.1 PBus コンポーネントによるスケジューラのリスト

```
/* PBus コンポーネント定義の
 プライオリティキュー ライブラリ
 関数内で排他制御を行っている */

/* プライオリティキューへの追加 */
extern void
myprioq_add(struct my_prioq *prioq,
   void *elem,
   unsigned long long priority);

/* プライオリティキューからの削除 */
extern void
myprioq_rem(struct my_prioq *prioq,
   void *elem);

/* プライオリティキューの先頭を取得 */
extern void *
myprioq_peek(struct my_prioq *prioq);

/* プライオリティキューの先頭を削除 */
extern void *
myprioq_pull(struct my_prioq *prioq);

/* プライオリティキュー */
struct myprioq prioq;

/* 独自属性の初期化処理の宣言 */
static inline int
pbus_thread_init_deadline(
 pbus_thread_deadline_t *deadlinep) {
 *deadlinep = ULLONG_MAX;
}

/* 独自属性の宣言 */
PBUS_FIELD(pbus_thread, /* 資源 */
   deadline, /* 属性 */
   unsigned long long);
    /* 属性の型 */

/* スケジューラへのスレッドの投入 */
static void
mysched_add(pbus_thread_t *t) {
 unsigned long long deadline;
 deadline =
 *pbus_thread_deadline(t);
 myprioq_add(prioq, t, deadline);
}

/* スケジューラからのスレッドの除外 */
static void
mysched_rem(pbus_thread_t *t) {
 myprioq_rem(prioq, t);
}

/* タイマ割り込み */
static void
mysched_tick(void) {
 /* キューの先頭のデッドラインが
  現在のスレッドより早ければ
  スケジューリングする */
 pbus_thread_t cur
 = pbus_thread_current();
 pbus_thread_t top;
 top = myprioq_peek(prioq, cur);
 if (top && *pbus_thread_deadline(top)
  < *pbus_thread_deadline(cur))
 pbus_thread_set_needresched(cur);
}
```

```
/* 次にスケジューリングすべき
 スレッドの選択 */
static pbus_thread_t
mysched_choose(void) {
 pbus_thread_t t
 = pbus_thread_current();
 if (pbus_state_running(t))
 mysched_add(t);
 return myprioq_pull(prioq);
}

static struct pbus_sched_ops
my_sched_ops = {
 .add = mysched_add;
 .rem = mysched_rem;
 .choose = mysched_choose;
 .tick = mysched_tick;
}

/* スケジューラの登録 */
static inline int mysched_setup(void) {
 pbus_sched_register(&my_sched_ops);
}

/* スケジューラの登録解除 */
static inline int mysched_cleanup(void) {
 pbus_sched_unregister(&my_sched_ops);
}

/* モジュールのコンストラクタと
 デストラクタの宣言 */
PBUS_MOD(sched_setup, sched_cleanup);
```