2008－OS－108 (3)
2008／4／23

カーネル機能拡張のための抽象化レイヤ P-Bus の実装

平野貴仁[†] 藤田肇[††]
松葉浩也[††] 石川裕[†,††]

"P-Bus" は, OS カーネルと拡張機能の間に入り, ソースコードレベルでカーネルに依存せずに機能拡張を行うことを可能にするための抽象化レイヤである. 本研究報告では, まず, 一般に, どのように型, 関数などのインタフェースを設計するべきかを説明し, 次に, スケジューラ変更について, 具体的なインタフェースの設計, Linux に向けたその実装について例証し, その成果及び課題について考察する.

Implementation of P-Bus, an Abstraction Layer to Extend Kernel Features

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

"P-Bus" is an abstraction layer between an OS kernel and extended features. It enables feature extansion not depending on the underlying kernels in the source code level. In this report, we describe how the interfaces such as types and functions should be designed and implemented, give an concrete example of the interface design and implementation for the scheduler replacement on Linux, and consider our achievements and issues.

1. はじめに

低価格, 低電力かつ高性能な CPU の普及により, 携帯電話, PDA, 自動車, ロボット, 家電等, いわゆる組込み機器の高機能化は顕著である. 高機能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, 新規デバイスへの対応などの機能拡張性, また瑕疵がないことの検証可能性が求められる. 我々は, このような機能拡張性や検証可能性を持ったオペレーティングシステムを実現すべく, ローダブルカーネルモジュール機構を発展させた P-Bus を提案した[1].

P-Bus は, Linux や, FreeBSD, Solaris など Unix 系 OS のカーネルに対してカーネルを機能拡張出来るように, カーネル資源に対する操作プリミティブを抽象化している. P-Bus 上に作られたカーネルモジュールは, P-Bus コンポーネントと呼ばれる. OS の機能拡張は P-Bus コンポーネントによって実現される. 現在, Linux 上で, P-Bus を実装している. P-Bus を FreeBSD や Solaris に移植することにより, P-Bus コンポーネントで実現された機能はこれら OS でも利用できるようになる.

P-Bus が提供する操作プリミティブのセマンティックスおよび P-Bus コンポーネントが満たさなければならない性質および状態を形式的に定義することにより, P-Bus コンポーネントを静的に検証できる枠組を提供する予定でいる.

本研究報告では, 第 2 節において, P-Bus インタフェースの全体設計について述べる. 第 3 節では, P-Bus を P-Bus オブジェクト群に分割して, スケジューリングに関連する P-Bus オブジェクトについて具体的なインタフェースの設計を挙げる. 第 5 節では, それらの Linux 上での実装について述べ, 第 6 節において, その成果及び課題について考察する.

2. P-Bus 設計の概要

Linux, FreeBSD, Solaris などの Unix 系 OS では, ローダブルカーネルモジュールにより OS カーネルのカーネル空間における機能拡張が可能である (図 1). しかしながら, 一般に, ローダブルモジュールが利用するカーネルのインタフェースには, 次のような問題がある.

OS 依存性 インタフェースがカーネルの種類や版ご

† 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 University of Tokyo

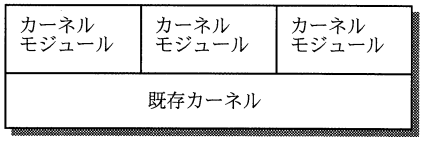

図 1 従来のカーネル機能拡張モデル

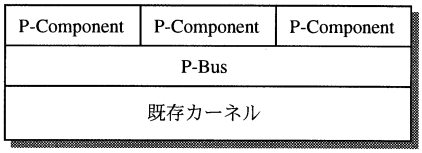

図 2 P-Bus モデル

とに異なっているため, 別の種類や版に対しては, ローダブルカーネルモジュールのソースを変更する必要がある.

曖昧性 インタフェースは, 明確に文書化されない傾向にあり, ローダブルカーネルモジュールの開発のために, カーネルのソースを理解する必要がある.

閉鎖性 スケジューラやメモリ管理といったコアな部分のアルゴリズムが変更できないなど, ローダブルカーネルモジュールの開発者にとって, インタフェースが十分に備わっているとはいえない.

提案する枠組みでは, カーネルとその機能拡張を切り分け, その間に新規な層である *P-Bus* を導入する. これにより, カーネル自体のインタフェースは変えることなく, 抽象化, 明確化, 柔軟化されたインタフェースを機能拡張を行いたいユーザに対して提供する. 実際に機能拡張を担当する部分を P-Bus コンポーネント (略称 *P-Component*) と呼ぶ (図 2). P-Component は, 従来のローダブルカーネルモジュールに相当する層である. P-Bus および P-Component の全ては, カーネル空間で動作する.

P-Component に対して OS 非依存なインタフェースを提供する P-Bus は, OS の種類や版ごとに変更しなければならない. したがって, P-Bus は, 最小限のコード量となるように, すなわち, OS のプリミティブなインタフェースだけを実現するように, 設計, 実装される.

P-Component の開発者は, P-Bus が提供するプリミティブを利用して機能拡張を行うことになる. この時, 他の P-Component でも利用できるような共通機能はライブラリとして利用できるようにすべきである. そこで, P-Component は別の P-Component を利用できる機構を提供する (図 3). 共有ライブラリといえる処理を独立させつつ, OS 非依存な層で実行することで, 全体としてのメンテナンスのコストを抑えることができると考えられる.

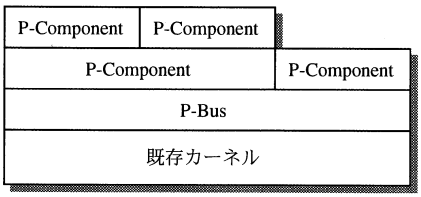

図 3 発展形 P-Bus モデル

2.1 P-Bus コアオブジェクト

P-Bus 層は, カーネルが提供する機能ごとに次のようなコアオブジェクトから構成される.

- プロセス, スレッド, メモリ管理
- スケジューラ, タイマ, CPU, 割り込み
- カーネル間通信, ネットワーク
- デバイスドライバ

P-Bus コアオブジェクト が提供するインタフェースは, 型, 関数, 定数, コールバック定義マクロに分けられる. P-Bus が公開する型, 関数, 定数, コールバック定義マクロの識別子は, `pbus_thread_t` のように, `pbus_` や `PBUS_` で始める. 一方, `schedule_pbus` のように, `_pbus` や `_PBUS` で終わる識別子や, `task_struct` のようなその他の識別子は, ヘッダファイルにおける閉じた利用のために定義されているもので, P-Component では, 利用してはならない.

2.2 型

OS 依存の各種リソースのハンドルやフラグの型は, P-Bus により, 抽象化され, 再定義される. 例えば, スレッドハンドルは, Linux では, `struct task_struct *` 型であり, FreeBSD では, `struct thread *` 型であるが, P-Bus によって いずれの OS でも `pbus_thread_t` 型となるように再定義される.

ここで, `pbus_thread_t *` 型に再定義するのではなく, `pbus_thread_t` 型としたのは, スレッドハンドルが一般にポインタである必然性はなく, 整数型や共用体型で定義されている可能性もあるためである.

P-Component は, OS 非依存とするためには, P-Bus で再定義された型を, たとえそれが実際にはポインタ型であったとしても, ポインタ型と見なしてその指す先にアクセスするなど, 特定の型であると見なして扱ってはならない.

2.3 関　　数

OS 依存の各種リソースに対する操作は, P-Bus によって抽象化され, 関数として提供される. ここでいう, リソースに対する操作とは, リソースに対応する構造体メンバの取得, 設定などの操作及び, リソース集合に対する追加, 削除, 列挙, ロックなどの操作などである.

こういった各操作のための関数に対して, その特性に基づいて, その関数内でスリープする可能性があるかどうかが規定されている

関数は, エラーが自明な一部関数を除き, 成功すると 0, 失敗すると正のエラーコードを返すように統一する. エラーコードは POSIX の定義をそのまま用いる.

なお, 関数は, 可能な限り, インライン関数で実現し, 関数呼び出しのオーバーヘッドは発生しないようにする.

2.4 定　　数

OS 依存の各種フラグやパラメータの値は, P-Bus によって抽象化される. P-Bus で定義される値と, OS で定義されている値は, 必ずしも一対一対応にできるとは限らないが, 一対一対応できた OS では, 変換するためのオーバーヘッドは発生しない.

2.5 コールバック定義マクロ

ローダブルカーネルモジュールの作成においては, あらかじめ, 関数ポインタ (群) を OS カーネルに対し登録しておき, あるイベントが発生した時に, それをコールバックさせる, という操作が必要となる. デバイスファイルが好例で, 関数ポインタを登録しておき, read, write, open, close などのイベントが発生した時に, コールバックを受け取ることにより, ファイルの操作を実現している. Linux のローダブルカーネルでは, デバイスファイルの操作を, 次のように記述する.

```
static ssize_t
my_read(struct file *file,
        char * __user *buf,
        size_t count, loff_t *ppos) {
   . ..
}
static const struct file_operations
my_fops = {
   . ..
   . read = my_read,
    . ..
};
```

P-Component では, これを, 次のように記述する.

```
static int
my_read(pbus_file_t file, void *buf,
         size_t count, size_t *count_out) {
   . ..
}
PBUS_FILE_HANDLER(my_fhandler,
                  . .., my_read,. ..);
```

ここで, PBUS_FILE_HANDLER が P-Bus の提供するコールバック定義マクロである. コールバック定義マクロは, 最初の引数がハンドラの識別子で, 残りの引数がハンドラに関連付けられるコールバック関数の識別子を示している. この例は, Linux でプリプロセッサを通すと次のように展開される.

```
static int
my_read(pbus_file_t file, void *buf,
        size_t count, size_t *count_out) {
   . ..
}
static int
my_fhandler_read_pbus(struct file *file,
                      char * __user *buf,
                      size_t count,
                      loff_t *ppos) {
    size_t count_out;
    int err;

    err = my_read(file, buf, count,
                  &count_out);
    if (!err)
        return -err;

    *ppos += count_out;
    return 0;

}
static const struct file_operations
my_fops = {
   . ..
   . read = my_fhandler_read_pbus,
   . ..
};
```

my_read が他の関数から呼ばれていなければ, コンパイラによりインライン展開が行われる. このように, マクロを利用することで, 関数呼び出し自体のオーバーヘッドを発生させずに, インタフェースの差異を吸収することができる.

それぞれのコールバック関数には, それぞれの特性に基づいて, 中でスリープ可能かどうかが規定されている.

3. 各 P-Bus オブジェクトの設計

ここでは, P-Bus オブジェクトのうち, P-Component においてスケジューリングアルゴリズムを作成するために必要なオブジェクトのインタフェースを紹介する.

3.1 P-Bus スレッドオブジェクト

P-Bus スレッドは, カーネルスケジューラが CPU 時間を割り振る実行単位である. スレッドモデルで N:M モデルなどを採用している OS の場合, アプリケーションから見えるスレッドにそのまま対応しているとは限らない. スレッドオブジェクトには, シグナルマスク, リアルタイムスケジューリング方針などを設定できる.

P-Bus スレッドの型は pbus_thread_t と定義される. スレッドオブジェクトでは, 次のような関数, コー

ルバック定義マクロが定義されている.

```
int pbus_thread_policy(pbus_thread_t thread);
int pbus_thread_prio(pbus_thread_t thread);
int pbus_thread_max_prio(int policy);
int pbus_thread_min_prio(int policy);

PBUS_THREAD_PRIO_LISTENER(
    listener,
    (void)(*pre_change)(pbus_thread_t thread),
    (void)(*post_change)(
        pbus_thread_t thread));
int pbus_add_thread_prio_listener(
    listener);
void pbus_remove_thread_prio_listener(
    listener);

int pbus_thread_set_needresched(void
    pbus_thread thread);

int pbus_add_thread_field(
    size_t size,
    void *data,
    pbus_thread_field_t *field);
void pbus_remove_thread_field(
    pbus_thread_field_t data);
void pbus_thread_field_data(
    pbus_thread_t *thread,
    pbus_thread_field_t *field);
```

pbus_thread_policy 関数は, スレッド thread で示されるスレッドのリアルタイムスケジューリング方針を返す. POSIX の sched_getscheduler 関数に対応するものである. ポリシは POSIX の定義をそのまま用いて, SCHED_FIFO, SCHED_RR, SCHED_OTHER, SCHED_BATCH のいずれかである. この関数内でスリープの可能性はない.

pbus_thread_priority 関数は, スレッド thread で示されるスレッドのリアルタイム優先度を返す. POSIX の sched_getparam 関数に対応するものである. この関数内でスリープの可能性はない.

pbus_thread_max_priority 関数, 及びその反対の pbus_thread_min_priority 関数は, それぞれ, リアルタイムスケジューリング方針 policy におけるリアルタイム優先度の最大値, 最小値を返す. POSIX の sched_get_priority_max, sched_sched_get_priority_min に対応するものである. これらの関数内でスリープの可能性はない.

pbus_add_thread_prio_listener 関数, 及びその反対の pbus_remove_thread_prio_listener 関数は, それぞれ, PBUS_THREAD_PRIO_LISTENER マクロを用いて宣言された listener を追加, 削除する. リアルタイムスケジューリング方針, 優先度が変更される際, 変更される直前に pre_change 関数, 変更された直後に post_change 関数が呼ばれる. pre_change 関数, post_change 関数の中では, スリープしてはならない. また, pre_change 関数が呼ばれてから post_change 関数が呼ばれるまでの間に, スリープすることはない.

pbus_thread_set_needresched 関数は, thread に対して, 今後 CPU の実行を明け渡すようにヒントを与える. これは割り込み等, その場では実行スレッドを切り替えることができないような場所で利用される.

pbus_add_thread_field 関数は, 各スレッドに対して, P-Component が自由に利用できる size バイトの data で初期化されたメモリ領域を追加し, field の指す場所にそのメモリ領域のハンドルを返す. メモリ領域のハンドルは pbus_thread_field_t 型である.

メモリ領域は, 存在するスレッドにすべてに追加され, また, 以後スレッドが開始されるごとに, 開始されたスレッドに追加される. また, スレッドが終了すると, 付随してこの領域も解放される.

もしもこのメモリ領域の確保が失敗した場合は, 既存のスレッドの場合は, pbus_add_thread_field 関数自体が失敗することで通知する. また新規に作成されるスレッドの場合は, 新規スレッドの作成が失敗することで通知される. この機構により, 失敗できない場所において, P-Component がスレッドに対するメモリ領域を確保しなければならない事態を防ぐことができる.

逆に, pbus_remove_thread_field 関数は, 既存のスレッドに対して確保されたメモリ領域すべて解放し, pbus_remove_thread_field 以前の状態に戻す. この関数内ではスリープする可能性がある.

これらの関数内ではスリープする可能性がある.

pbus_thread_field_data 関数は, thread の field で指し示されるメモリ領域を取得する. この関数内ではスリープする可能性はない.

3.2 CPU 集合オブジェクト

CPU 集合の型は pbus_cpus_t と定義される. CPU 集合オブジェクトでは, 次のような関数, コールバック定義マクロが定義されている.

```
const int PBUS_CPUS_SIZE;
int pbus_cur_cpu(void);

PBUS_CPUS_LISTENER(
    listener,
    int (*pre_up)(int cpu),
    void (*cancel_up)(int cpu),
    void (*post_up)(int cpu),
    int (*pre_down)(int cpu),
    void (*cancel_down)(int cpu),
    void (*post_down)(int cpu));
int pbus_add_cpus_listener(listner);
```

```
int pbus_remove_cpus_listener(listner);

int pbus_lock_cpus(void);
int pbus_unlock_cpus(void);
```

PBUS_CPUS_SIZE 定数は, CPU 番号のありうる最大値 + 1 を返し, pbus_current_cpu 関数は, 現在実行中の CPU 番号を返す.

電力節減や障害回避などのために, CPU を起動中に接続, 切断する機能が備わっている場合がある. pbus_add_cpus_listener, 及びその反対の pbus_remove_cpus_listener 関数は, それぞれ PBUS_CPUS_LISTENER マクロを用いて宣言された listener を追加, 削除する. CPU の追加処理の前に, pre_up コールバック関数が呼ばれ, 追加処理の後, 成功すると, post_up コールバック関数が, 失敗すると, cancel_up コールバック関数が呼ばれる. down_ 系コールバック関数についても同様である.

pbus_lock_cpus 及び pbus_unlock_cpus 関数は, CPU の接続, 切断を禁止, 解禁する関数である. これらは OS にかかわらずスリープ系ロックで実装されている.

3.3 P-Bus タイマオブジェクト

```
const int PBUS_TIMER_FREQ;

PBUS_TIMER_LISTENER(
    listener,
    void (*on_timer)(void),
int pbus_add_timer_listener(listener);
void pbus_remove_timer_listener(listener);
```

PBUS_TIMER_FREQ はタイマの周波数を返す.

pbus_add_timer_listener 関数, 及びその反対の pbus_remove_timer_listener 関数は, それぞれ, PBUS_TIMER_LISTENER マクロを用いて宣言された listener を追加, 削除する. タイマ割り込みごとに on_timer コールバック関数が呼ばれる.

3.4 P-Bus スケジューラオブジェクト

スケジューラは, 実行可能なスレッドのキューを管理する. 休止中のスレッドが実行待ちになったときに, 実行可能なスレッドのキューに追加したり, またスレッドが実行を中断したときに, 次に実行すべきスレッドを実行可能なスレッドのキューから選択したりする. そのキューからの選択, 追加するアルゴリズムは, P-Bus によって変更することができる.

```
pbus_thread_t pbus_sched_idle(int cpu);

PBUS_SCHED_HANDLER(
    handler,
    void (*add)(pbus_thread_t thread),
    pbus_thread_t (*replace)(
```

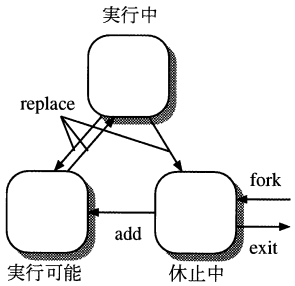

図 4 add と replace の位置付け

```
        pbus_thread_t thread, int add));
int pbus_select_sched_handler(handler);
```

pbus_sched_idle は, CPU 番号 cpu のアイドルスレッドを返す.

PBUS_SCHED_HANDLER では, 休止中のスレッドを実行可能なスレッドに加えるコールバック関数を, add, 現在実行中のスレッド thread を中断し, 次のスレッドを実行可能なスレッドから選び出すコールバック関数を replace として定義するものである (図 4). 現在実行中のスレッドを実行可能なスレッドに戻さない場合, すなわちスレッドがスリープする場合は, replace の引数 add には 0 が渡り, そうでない場合には非 0 が渡る. add と replace は, ともにスリープしてはならない.

pbus_select_sched_handler 関数は, すでに説明した PBUS_SCHED_HANDLER マクロで定義された handler にスケジューリングアルゴリズムを置換する. pbus_select_sched_handler 関数は, スリープする可能性がある.

4. P-Bus の実装

P-Component は, P-Bus のインタフェースを利用することで, ソースコードのレベルで, OS に依存することなく記述できる. ただし, プリプロセス, およびコンパイルの後は OS 依存のコードとなるようにして, P-Bus により性能が犠牲にならないように工夫する (図 5).

本節では, 各 P-Bus オブジェクトがどのように実装されているかについて述べる.

4.1 スレッドオブジェクト

Linux では, pbus_thread_t 型は, プロセスと同じく struct task_struct * 型にマッピングされる.

pbus_thread_policy 及び pbus_thread_priority 関数は, それぞれ Linux の struct task_struct 構造体の policy メンバ, rt_priority メンバの取得にそのまま対応している. pbus_thread_max_priority 関数, 及び pbus_thread_min_priority 関数は, そ

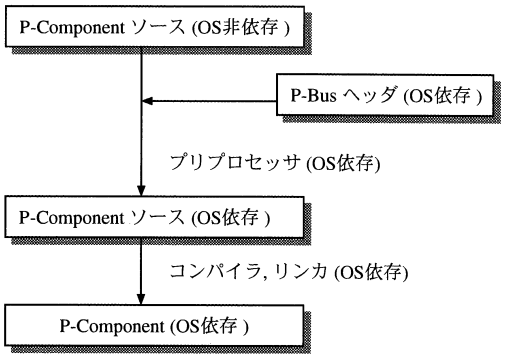

図 5 P-Component のビルドの流れ

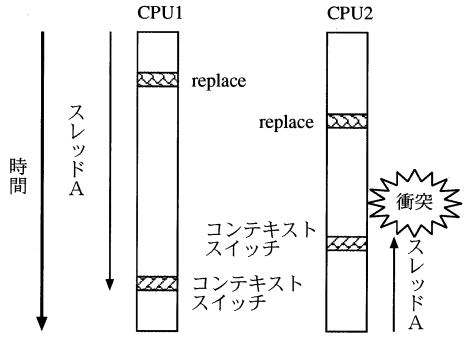

図 6 replace と実際のコンテキストスイッチのタイムラグ

れぞれ Linux の sys_sched_get_priority_max 及び sys_sched_get_priority_min に対応している.

また, Linux の setscheduler 関数がバイナリパッチされ, SCHED_THREAD_SCHED_LISTENER の pre_change, post_change コールバック関数を呼び出すように修正されている.

pbus_add_thread_field 関数, 及びその反対の pbus_remove_thread_field 関数は, Linux では, メモリキャッシュ機構を利用して実現される. 具体的には, メモリキャッシュを確保, 解放する関数であり, struct task_struct 構造体を確保, 解放するためにも使われる kmem_cache_alloc, kmem_cache_free 関数が, P-Bus によってバイナリパッチされる.

pbus_thread_set_needresched 関数は, Linux の struct thread_info 構造体の flags メンバへの_TIF_NEED_RESCHED の設定に対応する.

pbus_add_thread_field 関数が呼ばれた際の動作を説明する. まず, 自分以外の全てのスレッドの実行を止め, 全てのスレッドのプリエンプションの状態, 及び congestion_wqh ウエイトキューの状態を参照して, メモリ確保中である可能性があるスレッドが存在しないことを確認する. 仮に存在していた場合は, しばらく待機してから再試行する. 次に, メモリキャッシュの状態を走査し, 現在確保されている struct task_struct 構造体のそれぞれに対して, 追加フィールドのメモリを確保する. さらに, 今後 struct task_struct 構造体を確保するたびに, 追加フィールドも確保するように, データを登録する. pbus_remove_thread_field 関数が呼ばれた際もほぼ同様の動作となる.

4.2 CPU 集合オブジェクト

PBUS_CPUS_SIZE 定数は, Linux における NR_CPUS に, pbus_current_cpu 関数は, smp_processor_id 関数に対応する.

pbus_add_cpus_listener 関数, 及びその反対の pbus_remove_cpus_listener 関数は, それぞれ, Linux の register_cpu_notifier 関数, ならびに unregister_cpu_notifier 関数に対応する.

pbus_lock_cpus 関数, 及び pbus_unlock_cpus 関数は, それぞれ, Linux の lock_cpu_hotplug 関数, ならびに unlock_cpu_hotplug 関数に対応する.

4.3 タイマオブジェクト

また, Linux の scheduler_tick 関数がバイナリパッチされ, SCHED_TIMER_LISTENER の on_timer コールバック関数を呼び出すように修正されている.

4.4 スケジューラオブジェクト

Linux では, P-Bus により, wake_up 系関数および schedule 系関数がバイナリパッチされ, それぞれ中で PBUS_SCHED_HANDLER マクロで宣言された add, replace コールバック関数を呼び出すように修正される.

Linux の wake_up は, スレッドが, すでに実行可能であったり, 実行中であったり, あるいは他の CPU で wake_up している途中であったりしても, 適用可能であるが, P-Bus は当該スレッドがそのような状態かどうかのフラグを管理しており, そのような状態でない場合のみ P-Component の add 関数を呼び出す. したがって P-Component の開発者は, そのような状態のスレッドに対応しなくてよい. すなわち, スレッドが二重に追加される可能性を考えなくてよい.

schedule では, スレッドの状態を確認し, replace を呼び出し, 返されたスレッドに実行を移す. replace と実行のスイッチにタイムラグがあるため, 別の CPU でキューに戻されたスレッドが, まだ他の CPU で実行中の場合がある (図 6). その場合は, P-Bus は他の CPU での実行が終了するまでスピンしながら待機し, そのスレッドの実行を当該 CPU で再開する. したがって P-Component の開発者は, replace されたのち実際にスイッチするまでのスレッドの実行については考慮しなくてよい.

スレッドオブジェクトやスケジューラオブジェクトを使って P-Component でスケジューラを実装した例を 付録 A.1 で示す.

5. 関連研究

SPIN[2] では, マイクロカーネルに対して, 検証済みの拡張機能を特権モードで動作させる機構を加えたものである. マイクロカーネルの性能面での欠点を改善はしているものの, マイクロカーネルのサーバとの通信の複雑性を解消するものではない.

CAPELA[3] は, OS の拡張機能に対する, 保護優先から性能優先に跨る多段階の保護機構である. 保護機構のために NetBSD カーネルの機能をラップするインタフェースを提供しているが, マルチ OS カーネルに向けた機能の抽象化するものではない.

6. おわりに

本研究報告では, P-Bus の全体としてのインタフェースの設計方針を述べ, それに基づいた, スケジューリングアルゴリズムを作成するために必要なオブジェクトのインタフェースの設計, Linux に対する実装を紹介した. また, それを用いて, OS に依存せずにスケジューリングアルゴリズムを変更するコードを実際に作成できることを示した.

現在, 第 2. 節で示した, スケジューラ関連ではないオブジェクトに対しても設計, 実装を進めており, これらに関しても, コード例の作成や評価を行う予定である.

今後は, FreeBSD への移植, 既存カーネルの層を P-Component のデバッグ用のカーネルに変更することによる P-Component のデバッグ支援, また P-Bus に検証ツールへのアノテーションを記述することによる P-Component の容易な検証といった課題にも取り組んでいきたいと考えている.

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) 平野貴仁, 藤田肇, 松葉浩也, 石川裕：PBus: 柔軟なカーネル機能拡張のためのインタフェース, *2007-OS-106*, 情報処理学会, pp.63–70 (2007).
2) 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).
3) Kourai, K.: A Framework for Easily and Efficiently Extending Operating Systems, Master's thesis, The Graduate School of the University of Tokyo (1999).

付　録

A.1 P-Component によるスケジューラの例

```
#include <pbus/base.h>
#include <pbus/proc.h>
#include <pbus/sched.h>
#include <pcom/lock/lock.h>
#include <pcom/algorithm/list.h>

/* P-Component が利用する各スレッドに関連づけられるフィールド */
static pbus_thread_field_t edf_field;

/* フィールドのデータ構造 */
struct edf_data {
    pbus_thread_t thread;
    struct pcom_list iter;
    struct timeval deadline;
};
static struct edf_data init_edf_data;

/* 実行可能キュー, スピンロックは OS 非依存なので P-Component ライブラリで実装 */
static struct pcom_list head;
static struct pcom_spinlock spinlock;

static void
__edf_sched_add(pbus_thread_t thread) {
    struct pcom_list *iter;
    struct edf_data *edf_data;
    struct timeval *old_deadline,
        *new_deadline;

    /* スレッドに関連づけられたフィールドのデータを取得 */
    edf_data = pbus_thread_field_data(
        thread, edf_field);
    new_deadline = edf_data->deadline;

    /* 実行可能リストのイテレータを順にたどる */
    for (iter = head.next; iter != &head;
         iter = iter->next) {

        /* イテレータの指すデータを取得 */
        edf_data = pcom_container(
            iter, struct edf_data, iter);

        /* デッドラインがそれより遅ければループを抜ける */
```

```
        old_deadline = &edf_data->deadline;
        if (old_deadline->tv_sec
            > new_deadline->tv_sec)
         break;
        if (old_deadline->tv_sec
            == new_deadline->tv_sec
            && old_deadline->tv_usec
            > new_deadline->tv_usec)
         break;
        }
    }

    /* そのイテレータの手前にスレッドを加える */
    pcom_list_add_before(
        iter, &edf_data->iter);
}

static int
edf_sched_add(pbus_thread_t thread) {
    int ret;

    pcom_spinlock_lock(&spinlock);
    __edf_sched_add(thread);
    pcom_spinlock_unlock(&spinlock);
}

static int
__edf_sched_replace(pbus_thread_t thread,
                    int add) {
    struct edf_data *edf_data;
    pbus_thread_t idle;

    idle = pbus_sched_idle(
            pbus_cur_cpu(void);

    if (add && thread != idle)
        __edf_sched_add(thread);

    if (pcom_list_empty(&head))
        return idle;

    /* キューから最初のスレッドを取り出し返す */
    edf_data = pbus_container(
        head.next, struct edf_data, iter);
    pcom_list_remove(
        &head, &edf_data->iter);
    return edf_data->thread;
}

static int
edf_sched_replace(pbus_thread_t thread,
                  int add) {
    pcom_spinlock_lock(&spinlock);
    thread = __edf_sched_replace(thread,
                                 add);
    pcom_spinlock_unlock(&spinlock);
    return thread;
}

/* スケジューラ用のコールバック定義マクロ */
PBUS_SCHED_HANDLER(edf_sched_handler,
                   edf_sched_add,
                   edf_sched_replace);

static int edf_sched_setup(void) {
    int err;

    /* 各スレッドにフィールドを追加する */
    err = pbus_add_thread_field(
        sizeof(struct edf_data),
        &init_edf_data,
        &edf_field);
    if (err)
        goto pbus_add_thread_field_err;

    /* スケジューラの切替を実行する */
    err = pbus_select_sched(
        &edf_sched_handler);
    if (err)
        goto pbus_select_sched_err;

    return 0;

    pbus_select_sched_err:
        pbus_remove_thread_field(
            edf_field);

    pbus_add_thread_field_err:
        return err;
}

static void edf_sched_cleanup(void) {
    /* 各スレッドからフィールドを削除する */
    pbus_remove_thread_field(edf_field);
}

/* P-Component 用のコールバック定義マクロ */
PBUS_COMPONENT_HANDLER(edf_sched_setup,
                       edf_sched_cleanup);
```

ユーザ空間から deadline を変更する部分については, 省略した.