システムソフトウェアと
オペレーティング・システム 87－7
（２００１．６．２８）

組み込み用マイクロカーネルOS lambda

久住憲嗣 * 中西恒夫 † 福田晃 *
*九州大学 システム情報科学府 , †奈良先端科学技術大学院大学 情報科学研究科
nel@f.csce.kyushu-u.ac.jp, tun@is.aist-nara.ac.jp, fukuda@f.csce.kyushu-u.ac.jp

あらまし

組み込みシステムが大規模化している現在,組み込み用オペレーティングシステム(OS)の保守性や開発効率が重要である. そこで,本研究ではマイクロカーネル構成を採用した組み込みシステム向けOS Lambdaの設計・実装を行っている. マイクロカーネル構成はモノリシックカーネルに比べ開発効率はよいが,性能が悪く,メモリを余分に消費する傾向にある. この問題を解決するために,組み込みシステムでは出荷後にソフトウエアを変更することはまれである性質を利用し,実装時にはマイクロカーネル構成で実装を行い,出荷時にはモノリシックカーネルに自動変換を行い,性能の改善をはかる手法を提案する. この手法を用い,マイクロカーネルを設計・実装し性能測定を行った結果,本手法が有効であることを確認できた.

Design and Implementation of the Lambda μ-Kernel based Operating System for Embedded Systems

Kenji Hisazumi * Tsuneo Nakanishi † Akira Fukuda *
*Graduate School of Information Science and Electrical Engineering, Kyushu University
†Graduate School of Information Science, Nara Institute Science and Technology

Abstract

With large-scale of embedded systems, improvement of development efficiency is one of the most important problems. In this paper, we design and implement an embedded operating system, called the Lambda operating system, which improves the maintainability and development efficiency of the operating system. The Lambda operating system employs micro-kernel architecture, which allows the operating system to be easily designed. In addition, we propose a method to improve operating system performance by reconstructing it in implementation. With the method, Lambda is implemented as monolithic kernel. The method allows the operating system to be quickly developed and gives high performance. This paper also shows that the method is useful through implementing a prototype of Lambda and its performance evaluation.

1　はじめに

近年, 情報家電の登場などにより, 組み込みシステムが大規模化してきている. 大規模化するにつれ開発工数が増加する反面, 競争の激化などにより製品を短期間に設計・開発し, 市場に出すことが求められている. このため組み込みシステムの開発において, 開発工数の削減が必要不可欠であり, 開発効率の向上が重要な課題となっている. 組み込みシステムの開発効率を向上させるために, 組み込み用オペレーティングシステム (OS) の保守性や開発効率が重要であるにもかかわらず, この点に注目し OS を実装している例は少ない.

そこで, 本研究では,OS の保守性や開発効率を向上させるため, マイクロカーネル構成を採用した組み込み用 OS Lambda の設計, 実装を行っている [1][2]. しかしながら, マイクロカーネル構成ではプロセス間通信を使用することにより, 従来のモノリシック構成のカーネルよりも性能が悪いという欠点を持つ. これは, 文献 [3][4][5] などにより大幅に改善されてはいるが, 依然モノリシック構成のほうが性能がよい. さらに, 各機能をプロセスやスレッドに分割することにより, 各プロセス・スレッドごとにスタック, プロセス管理ブロックなどが必要なので, メモリを多く消費する傾向にある. これらの欠点は, 汎用 OS であれば問題にならないことも多いが, 組み込み用途のような資源を大幅に制限されたシステムでは致命的な欠陥になりうる.

そこで, 本研究はマイクロカーネル構成の各機能の独立性の良さを生かしつつ, 組み込みに特化し, 性能やメモリ消費量を改善することに重点を置く.

2　マイクロカール構成の問題点と解決策

マイクロカーネル構成では, 機能はプロセスとして実行されるので, 各機能の独立性が高く, 機能の組み替え, 追加, 削除が容易である. また, 一つの機能に不具合があった場合に, ほかの機能へ及ぼす影響が少ない. この性質は,OS やドライバを効率よく開発するのに非常に都合がよく,OS やドライバを開発する機会が多い組み込みシステムに適した性質である. しかしながら, モノリシック構成で OS を実装した場合に比べ, 性能が低く, また, メモリ消費量が大きい傾向にある.

この問題点を組み込みシステムに特化した方法で解決するために, ここで組み込みシステムの性質について考察する. 組み込みシステムでは, ソフトウエアが ROM に焼かれ出荷されるため, 出荷後にソフトウエアを変更することはまれであり, また困難でもある. さらに, 出荷後にソフトウエア構成を動的に変化させることもほとんどない. つまり, 出荷後にシステムを動作させながら拡張したり, 変更したりする必要はない.

そこで, 実装時には各機能を通常プロセスとして実装・デバッグを行い, 出荷時にはカーネル内プロセスとして実行し, 性能の向上を図れる. 通常プロセスモードとカーネル内プロセスモードは実行時に選択でき, 実装したソースコードに手を加える必要がないようにカーネルを実装する. この機能を利用することにより, アプリケーションやミドルウエアをカーネル内に組み込み, 性能の向上を図ることが容易になる [6].

しかし, プロセスをカーネル内に配置しても, 他プロセスの機能を呼び出す際にはプロセス間通信を行う必要あり, 多くの組み込みシステムのように OS, ミドルウエア, アプリケーションを一枚岩に実装した際よりも性能が低下する. そこで, この性能低下の原因であるプロセス間通信を自動的にできるだ

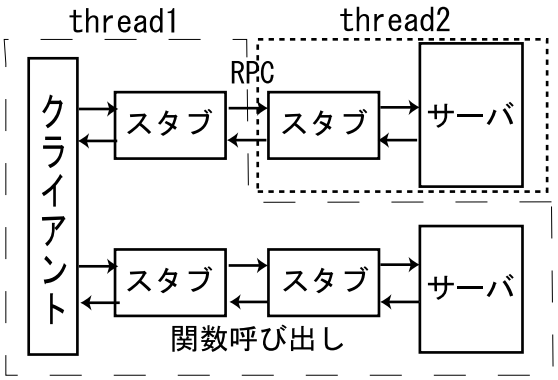

図 1: サーバスタブとクライアントスタブ

け排除し,関数呼び出し等に置き換えることにより,性能の向上を図る. 関数呼び出しに置き換えられたシステムは,すでにマイクロカーネル構成ではなく,モノリシックカーネルなので,この手法をモノリシック化と呼ぶ.

モノリシック化することにより,各機能が静的にリンクされるので,最適化が可能になる利点もある.

3 モノリシック化手法

Lambda では遠隔手続き呼び出し (RPC) を頻繁に使用するので,RPC に限定し,モノリシック化手法について述べる.

RPC を使用する際には,スタブと呼ばれる小さなコードを使用する. クライアント側スタブはクライアントから呼ばれた際に,引数をコード化し, サーバにリクエストを伝える役目を果たす. そして, サーバ側のスタブがそのリクエストを受け,引数を復号し, サーバに渡して処理を行う.

このスタブに工夫し,クライアントとサーバが同じプロセス内で動作している場合には関数呼び出しを行い,違うプロセスの場合には通常通り RPC を行うように実装を行う (図1). この手法を使用することにより,RPC を関数呼び出しに置き換えることができる. 詳しい実装については,第 7 節で述べる.

4 組み込み用 OS Lambda の設計方針

組み込み用 OS Lambda を実装するに際して, 以下の項目を設計方針とする.

- 多様な組み込みシステムへの適用
  組み込みシステムは様々なハードウエアを用いる. しかも, アプリケーションプログラムは特定用途の物が多い. そのため, 様々なハードウエアやアプリケーションプログラムに無駄なく適応できる必要がある.

- OS やドライバの開発効率の向上
  組み込みシステムでは特殊なデバイスを用いることが多く, とくにドライバの開発効率が重要視される. また, アプリケーション用に OS に特殊な機能を追加する機会も多い. そこで, 開発効率の向上が必要となる.

- マイクロカーネルの機能の削減
  マイクロカーネルの機能をできるだけ削減すると, それ以外の OS の機能をユーザプロセスとして実装できるようになり,OS の機能を変更し, デバッグすることが容易になる.

- プロセス間通信の性能の向上
  単層カーネル構成の欠点である性能の低下を, 最小限にとどめるため, プロセス間通信の性能の向上に多大な努力を払う必要がある.

5 システムの構成

組み込み用 OS Lambda は以下のモジュールから構成される (図2).

- マイクロカーネル
- メモリ管理サーバ

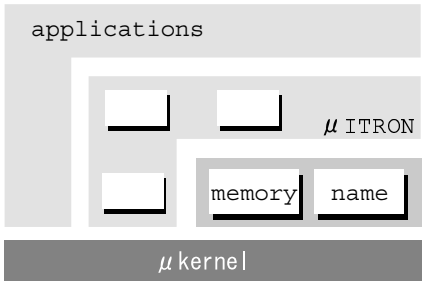

図 2: Lambda の構成

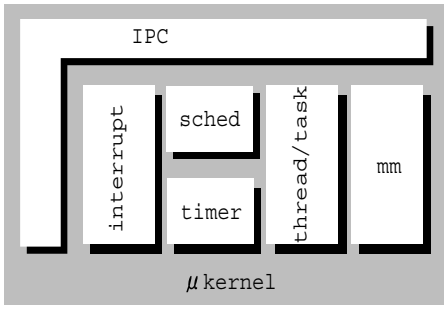

図 3: μ-kernel の構成

- 名前管理サーバ
- μ-ITRON API サーバ

これらを補助する物として,RPC の利便性を向上させ, 性能を向上させるために, Lambda Interface Generator(LIG) をサポートする.

今回は特に実装が進んでいるマイクロカーネルについて説明する.

6 マイクロカーネル

マイクロカーネルは以下のモジュールから構成される (図 3).

- スレッド・プロセス管理機構
- 割り込み管理機構
- タイマドライバ
- メモリ管理機構
- プロセス間通信
- スケジューラ

6.1 スレッド・プロセスモデル

Lambda のプロセスは Mach 等のように保護と資源管理の単位であり, 1 つ以上のスレッドを持つ. このプロセスはユーザの指定により, カーネル空間内で動作させることがあるが, その場合の保護は当然, 有効ではなく, 注意が必要である.

スレッドは実行実体であり, 1 つのプロセスまたはカーネルに属する.

コンテキストの保存・回復はハードウエア抽象化層として独立して実装する. また, カーネル内には単純なスケジューラを実装したスケジューラモジュールの組み込みを行う. 複雑なスケジューリングをおこなう場合には, スケジューラサーバの実装を行い, カーネル内のスケジューラモジュールと協調動作することによって, 複雑なスケジューリングの実現を行う.

6.2 割り込み管理機構

ドライバ等の割り込みを受信するスレッドは, 割り込み要因に対応した割り込み受信要求をカーネルに出す. 割り込みが発生すると, マイクロカーネルは割り込み要因に対応したポートにプロセス間通信で通知する. そして, その通知を割り込み処理スレッドが受け取とり, 割り込みを処理する.

割り込み管理機構はハードウエア抽象化層と, 割り込み通知部に分かれる. 特に応答性能を要求する時のために, ハードウエア抽象化層に直接割り込みハンドラを登録し, 割り込みを処理する方法も提供するが, 移植性が無くなり, また, 制約が大きい.

6.3 プロセス間通信

Lambda は以下のような特徴を持つ単純なプロセス間通信をサポートする.

- 同期式
- 一回のシステムコール発行で同時に送信と受信を行う

また, プロセス間通信を使用する際のデータの受け渡し方法として以下の二つをサポートする.

- データのコピー
  送信側の指定した値を, 受信側の変数にコピーすることによるデータ転送である. 実装する際はできる限りレジスタを使用してデータの転送を行うことにより, 高速化を行う. また, アドレス空間をまたがったデータのコピーを行う際は, 送信側データからカーネル空間にデータをコピーし, さらに受信側にデータをコピーする必要があるが,Lambda ではカーネル空間へデータをコピーしそのページを受信側にマッピングすることによりデータの転送量を減らす.
- 送信側アドレス空間へのマッピィング
  送信側の指定した領域を, 受信側のアドレス空間にマッピングすることによるデータ転送である. メモリ領域を転送する際に, すべてをコピーする方法の方が柔軟性はあるが, 非常に低速となる. そこで, 指定したページを送信側のアドレス空間に直接マップすることにより, 転送のためのコストを削減する.

マイクロカーネル構成の OS では, プロセス間通信の性能がシステム全体の性能を左右するので, できる限り単純かつ高速に実装する.

プロセス間通信は, 割り込み通知, ページフォルトの通知にも使用され, その場合の送信元はカーネルとなる.

6.4 メモリ管理機構

Lambda は仮想メモリの機能を使用し, メモリ保護をおこなう. 1 つのカーネルアドレス空間, 複数のユーザプロセス空間をもつ.

メモリ管理は, ページ管理テーブルの指定等, ハードウエアに直接依存する部分はハードウエア抽象化層としてマイクロカーネル内に実装を行うが, そのほかの物理メモリの割り当て管理等はメモリ管理サーバとして実装を行う. カーネルに実装が必要な機能としては以下のものがある.

- ページ管理テーブルを作成し, 初期化を行う.
- 指定されたページ管理テーブルをプロセッサに指定する.
- 指定された物理メモリを仮想メモリにマップする.

システム起動時には, 物理メモリはすべてメモリ管理サーバに割り当てられ, プロセス間通信のメモリマップの機能を利用することにより, 他のプロセスへの物理メモリ割り当てを行う. ページフォルトはプロセス間通信によりカーネルから通知される. 当然ながら, メモリ管理サーバは常に物理メモリに配置する必要がある.

メモリマップド IO も同様の方針で管理を行う.

7 LIG

カーネルがサポートする RPC を, 手続き呼び出しのように使用可能とするため, また可能であれば RPC を通常の手続き呼び出しに置換するために, インターフェース生成器 LIG(Lambda Interface Generator) を提供する. LIG を使用したモジュールの開発は以下のようになる (図 4).

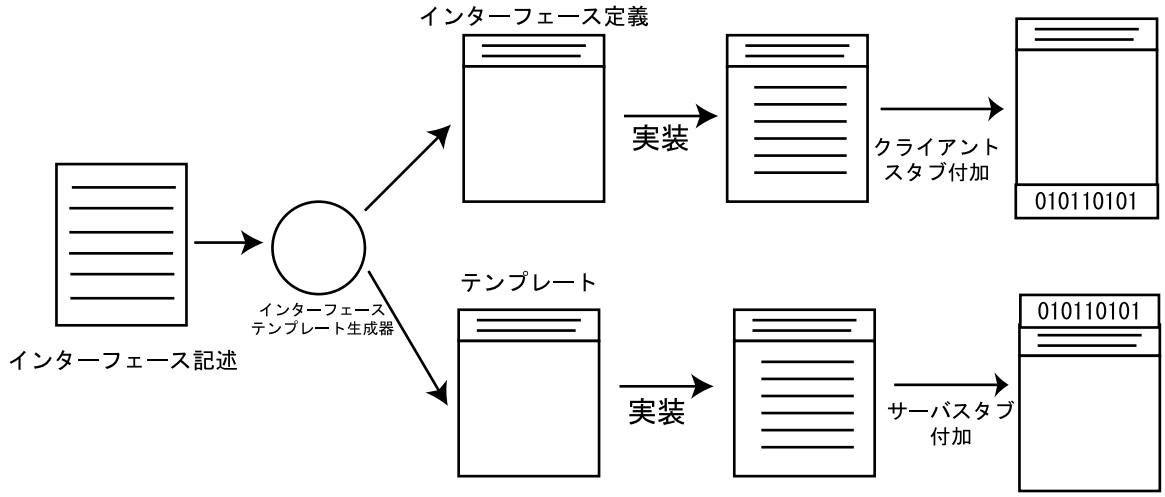

図 4: 開発の流れ

1. 下のような文法を持つインターフェース記述言語で定義する.提供したいサービスのインターフェースを定義する.

```
int rpc_call(out int arg1,
             in  int arg2);
```

2. LIG は,そのインターフェース定義を元に下記のようなクライアントが使用するためのプロトタイプ宣言を生成する.

```
int rpc_call(int arg1,
             int *arg2);
```

3. クライアント実装者は,このプロトタイプ宣言を元にクライアントの実装を行う.

4. また,同時に下記のようなサーバが実装すべき手続きのプロトタイプ宣言を生成する.

```
int rpc_call_impl(int arg1,
                  int *arg2);
```

5. サーバ実装者はこのプロトタイプ宣言を元に,サーバ手続きの実装を行う.

6. LIG はクライアントのコンパイル時にクライアント用のスタブを生成する.プロセス構造定義により,RPC を行うスタブを生成するか,手続き呼び出しを行うスタブを生成するかを指定する.

```
/* 同じアドレス空間に配置されていて
   関数呼び出しに置き換える場合 */
int rpc_call(int arg1,
             int *arg2)
{
   return rpc_call_stub (....);
}

/* 違うアドレス空間に配置されていて
   RPC を用いる場合 */
int rpc_call(int arg1, int *arg2)
{
   int retval;
   call (....);
   return retval;
}
```

7. LIG はサーバのコンパイル時に,RPC サーバ, 手続き呼び出しスタブの両者を生成する. しかし, プロセス構造定義により,RPC サーバの削除を指定した場合は, 手続き呼び出しスタブのみを生成する.

```
void rpc_call_server ()
{
   int arg1, arg2, retval;

   recv (....)
   for (;;)
   {
      ret = rpc_call_impl (....);
      send_reply_and_recv (....);
   }
}

int rpc_call_stub (int arg1,
                   int *arg2)
{
   return rpc_call_impl (....);
}
```

このようにして, 必要のない RPC を関数呼び出しに置換し, 性能の向上を図る.

8 予備評価

マイクロカーネル構成での性能の評価のために,本研究で実装したカーネルにおいて, 関数呼び出し,RPC,RPC をモノリシック化した物を通信手段とした機能呼び出しの性能の測定を行った. ターゲットは SH-3 80MHz である.

表 1 は, 各種機能呼び出し手段を用いて手続きを実行し, その返り値を得るまでの時間を示す. 呼ばれる手続きにはカウンタをインクリメントするだけの単純な物を用意した.

表 1: 性能測定

機能呼び出し手段	所要時間 (μsec)
RPC	50
モノリシック化 RPC	1.1
関数呼び出し	0.7

表 1 の結果から, モノリシック化 RPC は通常の関数呼び出しほどではないが,RPC から比べたら十分に高速に機能の呼び出しを行えることがわかる. これにより,RPC をモノリシック化する事により,OS の性能を大幅に向上させることができる.

9 まとめ

本研究では,OS の保守性や開発効率を向上させるため, マイクロカーネル構成を採用した組み込み用 OS Lambda の実装を行う. そのマイクロカーネル OS 用のプロセス間通信を関数呼び出しに置換するツールを実装し, 性能の予備評価を行った. RPC, 関数呼び出しに置き換えた RPC, 関数呼び出しの 3 種類について性能測定を行った結果, 関数呼び出しに置き換えた RPC は, 関数呼び出しに比べて多少のオーバーヘッドがあるものの,RPC よりも大幅に高速に実行でき本手法が有効であることを確認できた.

謝辞

本研究の一部は科学技術振興事業団戦略的基礎研究推進事業 (CREST) の支援のもとに行われたものである.

参考文献

[1] 福田晃, 最所圭三, 片山徹郎, 中西恒夫: 組込システム向け実行環境の自動生成

– δプロジェクトの構想 –, 電子情報通信学会技術研究報告, No.726, CPSY 99-125, pp 17–22, 2000.

[2] 久住憲嗣, 森若和雄, 中西恒夫, 片山徹朗, 最所圭三, 福田晃: 組み込み用マイクロカーネル OS の設計,SWEST2 予稿集, pp 49–53, 2000.

[3] Jochen Liedtke: Improving IPC by Kernel Design, Proc 14th SOSP, pp 203–205, 1993.

[4] Jochen Liedtke: Achieved IPC Performance, Proc 6th Workshop on HotOS, pp 28–31, 1997.

[5] Jochen Liedtke: On μ-Kernel Construction, Proc 15th SOSP, pp237–250, 1994.

[6] Jay Lepreau: In-Kernel Servers on Mach 3.0: Implementation and Performance, Proc USENIX Mach III Symposium, pp39–56, 1993.

[7] 田中義照, 川口 浩, 最所 圭三, 福田 晃, 組み込みシステム向けオペレーティングシステムの構成, 電子情報通信学会 CPSY 研究会, Vol.98, No.686, pp39-46, 1999.

[8] 坂村健: μITORN3.0 標準ハンドブック, 1997.

[9] 日立製作所: 日立 SuperH RISC Engine SH7708 シリーズハードウエアマニュアル, http://www.hitachi.co.jp/Sicd/Japanese/Products/micom/shmicom.htm