OS のサービス提供方式に関する研究

慶應義塾大学環境情報学部徳田研究室

杉野 洋一
53++@sfc.wide.ad.jp

指導教授
徳田英幸
村井純
楠本博之
中村修

1999 年 2 月 3 日

概略

本研究は, オペレーティングシステム (以下 OS) のサービスをカーネル (主記憶常駐部分), サーバ, ライブラリ内で提供する方式に分類し, 各提供方式の比較をし, その特徴を明らかにする. その為に, まず, 各提供方式の各方式の定性的な利点, 限界について整理し, 次に時間費用の観点からこれを評価する.

OS が提供するサービスの配置方式について, サービスを受ける実行主体の記憶空間を中心に三つに分けることができる. 一つはその実行主体の記憶空間, 一つはその実行主体とは別の実行主体の記憶空間, 一つはカーネルの記憶空間である. これは, カーネル内, ライブラリ内, サーバ内の三つとして考えることもできる. 伝統的な OS の機構では, サービスはカーネル内にシステムコールという形で実装される. しかし, この方法では時間費用と拡張性において問題がある事が指摘されている. 又, その為の代替案として, ライブラリ及びサーバによるサービス提供が提案されている. しかし, これら三つのサービス提供方式について, 利点, 限界, 費用を含む特徴については未整理である.

本研究では, まず, 既存の各 OS に関する文献調査から, 各提供方式の定性的評価を明らかにし, その上で, 定量的評価を定める為に時間費用を測定した. 時間費用の評価として, 実際の処理は何もしない空の呼び出しと, シリアルのレジスタへ読み書きを行う呼び出しに関する測定を行った.

各サービスの時間費用と定性的評価が明らかになることにより, OS にあるサービスを追加する場合, その為に最適な方式が明らかになる. この事から, 三つの提供方式から的確な選択が可能となる.

本論文では, 第2章にて, 拡張可能な OS の必要性について簡単に触れると共に, 単層カーネルの問題点について述べる. 第3章では, 三つのモデルの原理と実装例について触れ, 第4章では, 測定の詳細を述べる. 第5章で結論を述べ本論文を締めくくる.

Abstract

This reserch compares the methodologies of Operating System (OS) services that are provided "in-kernel", "in-library" and "in-server". By this reserch, in case of adding a service to OS, the most suited methodology is easily found.

For that, first, it mesure each overhead. And next it consider each features and limitation. In recent time, the three ways that are in-kernel and in-library and in-server are suggested. In a traditional OS architecture, services are concentrated in kernel as system calls. However this way is pointed out that it have some problems of extesivility and overhead. As substitutions, in-library and in-server services is suggested. Still their feature, limitation, cost are not well-orderd. The time costs is evaluate by round-trip time mesurements of two functions, a function that have a argument as integer and return a integer and a function that argument a pointer and substitute a physical device register values for the pointer's context, on RT-Mach and FreeBSD each. The features and limitations is evaluated by literatures.

The structure of this paper is as follows: Chapter2 describe some problems of a monorithic kernel, Chapter3 outlines 3 models of service placements, and consider the features of models except for the time costs, Chapter4 explain the measurements in detail, and Capter5 concluse.

目次

図目次

表目次

第 1 章

序論

1.1 はじめに

ハードウェア技術の発展は, 計算機が処理する対象を文書から連続メディアなどに多様化させた。又, PDA (Personal Digital Assistant) や NC (Network Computer) 等, ハードウェア基盤も同様に多様化している. これら新機能の追加の為, 拡張可能な OS について多くの議論がなされている. より拡張性の高い OS の実現の為には, 伝統的な OS の機構である単層 (モノリシック) カーネルでは限界がある. その為に, マイクロカーネル [1] や LibOS[2] と言った機構が提案されている.

マイクロカーネルではサーバを利用し, LibOS ではライブラリ上で OS のサービスが実現されている. 又, カーネルサービスにおいても, LKM[3] やカーネル内スレッド [4] の利用により, 従来, 問題の多かったカーネルサービスを改善する研究がある.

拡張性の高い OS の実現により, サービスを予めカーネル内に保持する事無く, 計算機を一旦停止しないで OS の機能を変化させる動的適応性や, よりアプリケーションに特化したサービスを実現する事ができる. 又, 多様化しているハードウェア基盤に容易に適応する事が可能となる.

本研究において OS とは, 複数の実行主体を動作させる為の, ソフトウェアの集合という意味で用い, 又, OS のサービスとは, 周辺機器多重化の為の資源管理, 及び資源提供方法 (抽象化) を指すものとする.

1.2 本研究の目的

本研究の目的は, OS のサービス提供方式を, サービスを受ける側の主記憶アドレス空間をもとに分類し, その利点と欠点及び限界について論じ, それらの特徴について比較し, これを明らかにすることにある. これにより, サービスが特定される時、それに最適な方式が明らかになるようにすることである.

伝統的な OS の機構である単層カーネルの問題点は, OS の全機能が予めカーネルに組み込まれている事である. 従って, 保守, 拡張作業の際に必要となるカーネル内の依存関係の把握, 新機能の動的追加, アプリケーションに特化したサービスの提供が困難である. 例えば, 計算機の移動による利用者の要求の変化は，要求される可能性のある全ての機能をカーネルが保持する事によって対応している．豊富な主記憶, 大容量の二次記憶, 或は高速な CPU 等，充分な資源のある計算機では，OS が全ての機能を内部的に保持する構造も可能である．しかし，資源が制限されている計算機では，要求され得る全ての機能を保持する事は難しく，主記憶, 二次記憶などの制限から不可能な事さえある．

又，要求される全ての機能をカーネルが保持する事は，依存関係の把握を困難にし, OS を複雑化させ, 保守, 拡張作業を困難にするなどの欠点がある．これは, ハードウェア基盤の多様化に対して, OS の適用領域を狭め, 開発効率を低下させる.

これら単層カーネルの問題点を解決する為に, 様々な方式が提案されている. しかし, 問題は個々の方式の善し悪しではなく, あるサービスがその特性に見合った方式で提供されているかである. つまり, あらゆるサービスに最適な提供方式を議論するのではなく, サービスの性質が特定されれば, そのサービスに最適な提供方式が明確になるように整理されていることが重要である. これにより, あるサービスを追加する場合に, そのサービスに最適な方法を特定できるようになる.

本研究では, 各提供方式の特徴を評価する為に, まず, サービス提供方式をサービスを受ける実行主体のアドレス空間をもとに三つに分類する. 即ち, 提供方式を, (1) サービスを受ける実行主体と同一のアドレス空間に配置する場合, (2) サービスを受ける実行主体とは別の実行主体のアドレス空間に配置する場合, (3) カーネルのアドレス空間に配置する場合の三つに分類し, これをそれぞれ, “Service by Library”, “Service by Server”, “Service by Kernel” と名付ける.

次に, 各方式の定性的評価を明らかにする為に, 広く文献を調査し, 実装例を上記の三つの分類に適用し, 詳細な分析を行い, それらの利点, 欠点, 限界点について考察する. さらに, 各方式の定量的評価をする為に, これら三つのサービス提供方式について, FreeBSD[5] と RT-Mach[6] を用いて, 整数を引数として整数を返すだけの関数とポインタを引数としてそのポインタに物理機器のレジスタの内容を代入する関数を, カーネル内, ライブラリ内, サーバ内に実装し, その呼び出しから返り値を受け取るまでの時間を測定する. 又, その結果を基に考察を行う.

これにより, 各提供方式の定性的或は時間費用の面からみた評価が定まり, それぞれの提供方式の特徴が明らかになる.

1.3　本論文の構成

本論文では, 第2章において, 拡張可能なOSが必要である理由について簡単に触れると共に, 単層カーネルの問題点について述べる. 第3章では, サービスの提供方式を三つのモデルに分類し, 原理と実装例について触れ, 第4章では, 測定の詳細と考察を述べる. 第5章で結論を述べる.

第 2 章

単層カーネルの問題点

本章では, 本研究で対象とする OS の定義を行い, 拡張可能な OS の必要性を示した上で, 単層カーネルの問題点を明らかにする.

2.1 OS の基本モデル

本研究で扱う OS とは, 複数の実行主体[1]を動作させる為に, 機器多重化と資源抽象化を扱う [2] ソフトウェアの集合である. 伝統的 OS では, これらの機能のほとんどがカーネルに集中していた. しかし, これらの機能の全てが必ずしもカーネルに存在する必然的理由は無い [1][2].

OS の基本的役割は二点ある.

一つは機器の多重化である. OS は通常, 複数の実行主体を動作させる為, 主記憶に常駐するカーネルを持っている. カーネルが計算機の資源を占有し, 提供している. カーネルが資源を一旦占有する事で, 複数の実行主体間の資源[2]に対する競合の防止や, 排他制御, 行き詰まり (dead lock) の防止等を実現している.

もう一つは, 抽象化である. OS はプログラマの便宜を計る為に, 個々の物理機器の特徴を隠蔽し, 資源を抽象化して提供している. 例えば, 二次記憶として固定ディスクを利用する場合, シリンダやセクタ, ヘッド数を用いてブロック単位でプログラムをするのは, データをファイルとして扱うのに比べて労力を要する. 更に Unix では周辺機器の一部もファイルとして参照できる. 一例として FreeBSD の /dev/stderr が挙げられる. これは標準エラー出力をファイルとして参照できるというものである. 又, Mach の “Process Port” は, 本来, IPC の為に利用されるポートを, 各スレッドのカーネルとの通信に用いている. この為, カーネルサービスの幾つかもポートとして参照できる. これも抽象化の一例である.

---

[1] 実行主体とは, 何らかの時間配分方針に応じて CPU 時間を割り振られるものを言う.

[2] 資源とは主記憶や CPU 時間, その他, 周辺機器等, 複数の実行主体に利用されるものを指す.

伝統的 OS においてサービスのほとんどは, カーネル呼び出しという形で実現されていた. この様にサービスの多くが, カーネル内で実現され, それらのサービスの動的な追加, 削除, 置換ができないカーネル及びその機構を単層カーネルと言う. サービスをカーネル内に保持する手法は, 次の三点で問題がある.

- 動的適応性の欠如
- 拡張性の欠如
- 特化性の欠如

2.2 新しい機構の基本的要件と単層カーネル

本節では, 新しい機構の基本的要件を挙げると共に, それらの要件を単層カーネルが満たせない理由について述べる.

2.2.1 動的適応性

OS における動的適応とは, 計算機を停止すること無く，アプリケーションの要求の変化に対応したサービスを提供可能とする事である．

動的適応機能の実現の為には，ハードウェアだけでなくソフトウェアにおいてもこの機能が実装されている必要がある．ハードウェアの小型化に伴って，計算機の携帯が可能となった．この様な計算機を利用する環境では，利用者の要求も動的に変化する．例えば，情報通信の帯域や電源管理の方針等が計算機の稼働中に変化する．

従来の単層カーネルの OS では, 必要となる全ての機能は，予めカーネルの中に組み込まれていた [7]．計算機の移動による利用者の要求の変化に対して，要求され得る全ての機能をカーネルが保持する事によって対応していた．その為, 機能の新規追加に対して, カーネルの再コンパイル, 再起動が必要であった. 例えば, 移動計算機環境で電源が AC 電源から電池に変化し, APM(Advanced Power Management) を利用しようとする場合, APM のサービスが予めカーネル内に含まれているか, 或はサービスを追加したカーネルを作成した後, 再起動しなくてはならない.

一方で, 高性能で充分な資源のある計算機では，この様なカーネルが全ての機能を内部的に保持する構造も可能であると言う意見もある．しかし，例えば小型化の為に, 資源が制限されている計算機では，要求の変化に対する全ての機能を保持する事は難しく，主記憶, 二次記憶などの制限から不可能な事さえある．同時に, 豊富な資源を有している計算機の場合でも, 提供可能なサービスは OS 設計者によって, 予め予測されたもののみに留まる. 例えば, 無停止サービスを前提としているサーバにも, 新機能の追加の為に, 単層カーネルは再起動を要求する.

以上の観点から, 動的適応が必要であるが, 機能をカーネルが予め内部的に保持している様な機構では, これに対応することができない.

2.2.2 拡張性

要求される全ての機能をカーネルが内部的に保持する事は，OS を複雑化させ保守拡張作業を困難にする．

依存関係が複雑に絡み合った集合体では, 部分の置換, 削除, 追加が難しい [2]. この事は障害部分の特定や, 或は開発を困難にする. 例えば, 連続メディア処理の為の QOS(Quality Of Service) 管理機能や実時間性を要求するアプリケーションがある. これらの為の新機能追加が, 単層カーネルの機構では難しい.

ハードウェアの小型化は携帯可能な計算機と共に，多様なハードウェア基盤を作り出す．例えば IEEE1394[8] で想定されているように, 家電製品にもネットワーク機能が付加され，他の計算機或は他の家電製品との間で通信が可能となる．既に PDA (Personal Digital Assistant) や NC (Network Computer) 等の様な特殊なデバイスを持った計算機も開発されている．これらの計算機では, 従来想定されていたようなキーボードとマウスの組合せという入力装置を持たず, 特殊な装置で入力をする場合がある. これらの計算機上では資源の制約から OS を利用しない事も考えられるが，効率的なプログラム開発を行う為には，OS による資源の抽象化が必要である．PDA や NC を含む様々な装置や基盤への容易な OS の移植を実現する為にも, カーネル内に機能を集中する事は望ましくない.

以上の観点から, OS に拡張可能性が必要である. しかし, 単層カーネル機構は依存関係の複雑さから, OS の適用領域の拡大を妨げ, 開発効率の低下をもたらす.

2.2.3 特化性

単層カーネルでは, OS がアプリケーションから, 完全に独立している為, 以下に示すように個別のアプリケーションに特化したサービスを提供する事が困難である. 個別のアプリケーションに特化したサービスとは多重 API[3]の事を言う.

単層カーネル機構では, 全てのアプリケーションが, 同じ抽象化を利用しなくてはならない為, 抽象化によるオーバヘッドを避ける事ができない. 従って, 速度を要求するアプリケーションや, 特殊な機能を利用するアプリケーションの要求に答える事が困難である.

その一例としてデータベースを提供するアプリケーションが挙げられる [9]. この様なアプリケーションでは, 実行速度などの問題からハードディスクをファイル単位ではなく, ブロック単位で扱うかも知れない. しかし, 単層カーネルでは抽象度は常に固定されている為, ディスクをファイルとして扱う以外の方法は利用できず, 従って, そのオーバヘッドを避けられない.

更に, ハードウェア技術の発展は, 計算機の扱うデータを文書中心から連続メディアなどを含むものへ多様化させた. これと共にアプリケーションの OS に対する要求も多様化

---

[3] 多重 API とは, 同時に, 異った実行主体間で, 異った API を利用可能なことである.

した. この様な状況では, OS は特殊なサービスを提供する必要がある. 例えば, 連続メディアの為に周期性を提供する様な場合が挙げあれる. この機能を, カーネル内に単層的に組み込むことは可能だが, 前述の動的適応性や拡張性の点で問題が起こる.

2.3 単層カーネルの問題点とサービス提供方式

前述の通り, 単層カーネルの問題点は, 動的適応性, 拡張性, 特化性の欠如にある. こうした単層カーネルの問題点を解消する為に様々な機構が提案され, それぞれの提供方式の有用性が主張されている. しかし, 問題は個々の提供方式の是非ではなく, あるサービスに最適な提供方式が, サービスの特性から明らかになるかである. このことにより, OS 開発者は新規機能を追加する場合, その機能に最適な提供方式でこれを開発できる.

次章ではまず, サービス提供方式を分類, 整理し, それぞれの方式の, 利点・欠点・限界点を明らかにする.

第 3 章

サービス提供方式の基本的モデルとその定性的評価

本章では, 現在提案されている OS のサービス提供方式をその実現する場所から三つに分類し [7], それらの実装例について述べる. これらの実装例の多くは, 複数の方式を利用している.

3.1 アドレス空間によるサービスの分類

仮想記憶を用いる OS の場合, 実行主体多重化の為の機構は, その主記憶アドレス空間と密接に関わっている [4]. 例えば,UNIX ではある実行主体が別の実行主体, 或はカーネルのアドレス空間に違法にアクセスした場合, 違反行為をした実行主体にシグナルを送り, 通常, その実行を停止する. この機構により, 悪意のある実行主体や, 誤りを含んだ実行主体の事故から, 他の実行主体及びカーネルのアドレス空間上のデータは保護される.

従って, 仮想記憶を用いる OS の場合, アドレス空間の違いによりその権利関係に違いがある. 即ち, OS の提供するサービスを実現する場所として, どのアドレス空間を選択するかが, そのサービスの限界に大きく関係する. 例えば, 利用者空間は通常, カーネルを通さずに周辺機器にアクセスすることはできない. これは, 利用者空間から周辺機器に関するサービスを提供する際の, 限界の一つとなる. 或は, 今回測定で利用した IOPL のフラグも, 権利の判定はアドレス空間単位で行われる.

アドレス切替えはサービスの時間費用にも多く関係する [4]. 従って, 速度の面からもこれらの分類は有効である.

ある実行主体を中心に考えた場合, 主記憶上のアドレス空間は次の三つに分類される.

- 自らのアドレス空間
- カーネルのアドレス空間
- 他の実行主体のアドレス空間

本研究では, この分類を基に, 上から順に “Service by Library” “Service by Kernel” “Service by Server” という命名 [7] を採用する. 次節では, それぞれの提供方式の詳細について述べる.

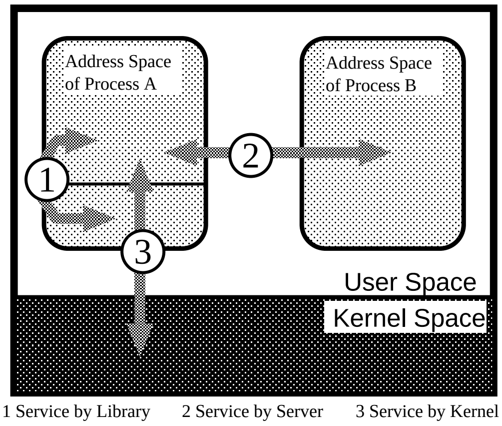

図 3.1: 3 モデルの概念図

図3.1は, 3 モデルの概念図を表す. “Process A” はサービスを利用する実行主体である. Process B” はサービスを提供する実行主体である. 図中の色の濃い部分はカーネルのアドレス空間を表現している. “1” は “Service by Library”, “2” は “Service by Server”, “3” は “Service by Kernel” を, ←→ は要求, 応答の相互作用を表す.

3.2 提供方式の詳細

本節では, それぞれの提供方式の詳細について述べると共に, それらの実装例について触れる.

3.2.1 Service by Kernel

“Service by Kernel” は, サービスを実現する部分がカーネルのアドレス空間に配置されている.

カーネル内でサービスを実現する事は, 従来, 動的適応や開発, 保守効率, 或はオーバヘッドなどの点で問題があった. その為, LKM (Loadable Kernel Module) [3][5][10][11] やカーネル内スレッド [4][12][13] を用いる方式が提案されている.

LKM では, 個々の実装に多少の違いはあるが, 拡張部品をカーネル空間内に置き, アプリケーションは拡張部品に定義されている手続きを呼び出すことによりこれを利用する. 個々の部品は, データ, 手続き或はその両方が定義され, 利用されない時には, 二次

記憶にファイルとして置かれ, 必要に応じて動的にカーネルのアドレス空間に取り込まれる. (FreeBSD と RT-Mach 上での具体的な定義方法については付録 B,Cに記載した.) 一旦, カーネル内に配置された部品は, 他のカーネルの部分と同等の権利をもつ. これに対して, 通常のカーネルのサービスは主記憶に常駐する. LKM の利用により, カーネルに動的に機能の新規追加をすることが可能となる.

一方で, 個々の部品としての LKM はカーネル空間に置かれるだけであり, 別途, 依存関係の把握や安全性の確保という面で他の機構, 例えば LKM サーバの補助を必要とする [3]. 又, LKM では正しいインターフェースを提供しているか, 外部シンボルを解決できるか等の点検をする事はできるが, 意味論的な誤りを検出する事はできない [3]. 例えば, “`while(1);`” の様な一文がモジュールの中に記述されていても, LKM 機構だけではこれを検知することができない.

カーネル内スレッドは, 通常, 横取り不可能な時間配分をされるカーネルモードで, CPU を横取り可能とすることによる仮想的な実行主体である. カーネルは, 通常の実行主体と異なり, ハードウェアからの割り込み或は実行主体からのカーネル呼び出しによって駆動する. カーネル内のコードを実行している際には, より優先度の高い割り込みが掛かった場合を除き, CPU を横取りされない. カーネル内スレッドでは通常のカーネル内のコードと異り, ある時間配分ポリシに基づき, CPU を横取りされる. カーネル内スレッドを利用することにより, カーネル内のデータに対するアクセスや, カーネル内の呼び出しを高速に行うことができる [4].

更に, カーネル内スレッドは, LKM や他のカーネル内の部分に比べて, 部品性が高い為, 権利関係に違いを持たせることが可能である. 例えば, Apertos[13] では, カーネルは MetaCore とそれ以外の複数のメタオブジェクトに分離され, 排他制御は MetaCore で集中して行われ, それ以外のメタオブジェクトでは行えない. これにより, カーネル内の排他制御が整理され, 拡張性が確保できると共に, 排他制御の集中管理により個々のスレッドの障害がカーネル全体に波及しない為, 安全性も増す.

カーネル内スレッドでは, 横取り可能方針で拡張部分が実行される為, 前述の “`while(1);`” の様なコードが記述されていても, システム全体の停止は防ぐことができる. しかし, この方法でも同一保護領域による安全性の低下, 或は障害孤立化をする為の上乗せ経費を避ける事はできない.

3.2.2 Service by Server

“Service by Server” では, サービスは, サービスを受ける側の実行主体とは, 別の実行主体のアドレス空間で実行される.

複数の実行主体間で, サービスの授受という関係がなりたつ時, サービスを提供する側をサーバと言う. サーバと, サービスを要求する実行主体の間では, 相互作用の為に, IPC(Inter-Process Communication) が用いられる. IPC とは, ある実行主体のアドレス空間から別の実行主体のアドレス空間へ安全にデータを転送する為の枠組である.

「安全に」とは, 相互作用をする両者に権利を与え, 権利以外の事を不可能にした状態である. 例えば, ある実行主体 A からある実行主体 B にデータを転送する時, A が B のアドレス空間の全てにアクセスできてしまっては, A が悪意のあるプログラムであったり誤りがあるプログラムであった場合には, B のデータは破壊されてしまう可能性がある. このような時には, A に B が用意したバッファのみにアクセスする権利を与えるか, 或は全権限をもつカーネルが A から B にデータをコピーするかして, それ以外のアクセスを防がなくてはならない.

この方式では, サービスは高い部品性 (modularity) を持つ [4]. 又, カーネルと別のアドレス空間を利用する為, サーバ上の障害がカーネルに波及しない. 従って, システムの耐故障性を向上させることが可能である. その一方で, この方式では速度が問題となる. IPC の速度がシステム全体の性能に大きく影響する [1]. サーバが何ら特権を持たず, 他の実行主体と同様の権利を持つとすると, サーバは資源を利用する為にカーネルを呼び出すことになる. このことも, “Service by Server” の時間費用を増大させる要因の一つである.

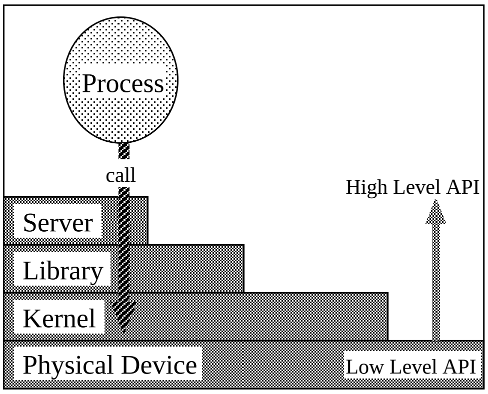

図 3.2: API の階層

図 3.2の例では, サービスが最終的に実現されるまでに三つの提供方式を利用している. つまり, サービスを受ける実行主体はサーバを呼び出し, サーバはライブラリを呼び出し, ライブラリはカーネルを呼び出している. システムの現在時間を取得するようなキャッシュの有効性が低いサービスでは, この様な階層化は余分な時間費用が掛かる. その為, 従来通りカーネルが物理機器へのアクセスを独占している場合, カーネル内サービスとしてこれらのサービスを実装するのが最速である. これは, “Service by Library” の場合も同様である.

機能の一部に, サーバを利用した OS として “ Mach ”[1] がある. Mach では, Unix の抽象化機能は, カーネルとしてではなく, サーバとして実装されている. サーバは, ユーザモードで走行し, 他の実行主体と同様に扱われる. Unix の提供する高度な機能, 即ち抽象化はユーザ空間に置かれ, Unix とのバイナリ互換を保っている. カーネル内には, 仮想記

憶, IPC, 実行主体の生成, 実行と破棄, デバイスドライバと言った機能が残っている. 他にも, “port” の名前空間の管理等がサーバとして実装されている.

3.2.3 Service by Library

“Service by Library” では, サービスは, サービスを受ける実行主体と同一のアドレス空間で実行される.

ライブラリによるサービスを実現した例として, Exokernel[2] や C.Maeda の実装 [14] がある. ライブラリは, 要求をした実行主体と同一のアドレス空間に配置され, 実行主体からの手続き呼び出しを用いて利用される.

Exokernel では, OS は抽象化と物理資源多重化を司る部分に分離され, 抽象化を扱う部分は libOS, 資源多重化を扱う部分はカーネルで実装されている. カーネルは, 物理資源を多重化し, それを安全に実行主体に対して提供する責任を負うのみである.

C.Maeda の実装は, Mach3.0 上で実装された TCP/IP,UDP/IP のプロトコルスタックである. send(2) や recv(2) と言ったシステムコールがライブラリ上で実現されている. この実装では, 接続の確立や接続の終了はカーネル内で実装している.

“Service by Library” では, アドレスの切替えを抑制する努力がなされ, 又, カーネルが複数の実行主体のキャッシュを一同に持つのに比べ, キャッシュの有効利用と言う観点 [2][15] からも高速である. キャッシュが有効な場合はライブラリ内でサービスが終了し, カーネル空間へのアクセスを極力減らす事ができる. 又, 有効なメモリ管理と言う点でも利点がある. 共有ライブラリはそれをリンクしている実行主体の終了と共に, 主記憶から排除される為, メモリを有効活用する事ができる. これは常駐型のカーネルやサーバに比べて大きな利点がある.

しかし, 複数の実行主体間のデータの共有及び転送と言う観点では慎重にこれを作る必要がある [9]. ライブラリそのものは, 特権を持つものではない為, 複数の実行主体間の相互作用の為には, 特権サーバやカーネルの助けが必要となる. 複数の実行主体間でデータを共有する際には, 排他制御が重要になる [12]. 実行主体に対して, 横取り可能な時間配分方針を取る OS では, CPU 時間を横取りされないのはカーネルのみである. 従って安全に排他制御を行う為にはカーネルの助けが必要である.

又, データの利用が断続的である場合, それらのデータはカーネル或はサーバ [2] が保持しているのが合理的である. FreeBSD[5] の /dev/lkm, RT-Mach[6] の /dev/timezone がその例である. この例では, いずれも物理機器を抽象化したものではない. FreeBSD の /dev/lkm は, カーネルがロードされている LKM の情報を, アプリケーションに提供する為のインタフェイスである. これを利用し, アプリケーションは LKM をロードしたり, アンロードすることができる.

これに対して, 個々のアプリケーションに特化したサービスを行うにはライブラリが適している. ライブラリは個々の実行主体に固有に付くものである為, 個々の実行主体に特化したサービスを提供しやすい.

ライブラリの種類として, 共有ライブラリと静的ライブラリが存在する. 共有ライブラリでは, ライブラリは動的にリンクされ複数の複数の実行主体に共有される. これに対して, 静的ライブラリはコンパイル時にリンクされ, 複数の実行主体に共有されない. 従って, 粒度の大きいサービスを, 静的ライブラリで実現すると, 主記憶, 二次記憶を占有する. 例えば, ファイルシステムの実現という様な粒度の大きい単位で, 一つのライブラリになっていることを考えると, それを静的ライブラリで実装すると記憶領域の無駄が大きい.

3.3 定性的評価のまとめ

表 3.1: 定性的評価のまとめ

	高速性	安全性	データ共有
LKM	○ (*)	×	○
In-kernel Thread	△	△	○
Service by Server	×	○	○
Shared Library	○ (*)	○	×
Static Library	○ (*)	○	×

表 3.1は定性的評価のまとめである. ○は他の方式と比較して良い, △は他の方式と比較して, 良い場合もあれば悪い場合もある, ×は他の方式と比較して適さないと言うことを表す. (*) 図 3.2で示したように, 物理機器にアクセスするか, 又はキャッシュの効かない様なサービスではカーネルサービスが高速であり, そうでないサービスではライブラリサービスが高速である.

3.4 本章のまとめ

本章では, OS のサービス提供方式を, サービスを利用する側から見たアドレス空間から三つに分類すると共に, その説明と実装例について述べた.

次章では, この分類をもとに, それぞれの方式の時間費用を明らかにする.

第 4 章

時間費用の観点から見た各サービスの評価

本章では, それぞれの提供方式の時間費用を明らかにする為に, 本研究で行なった測定及びその結果を述べる.

4.1 測定目的

本測定の目的は, 各提供方式の時間費用を明らかにすることによる, 各提供方式の定量的評価である. サービスの安定性や, 高速性を必要とするサービスも存在する為, 各提供方式の時間費用が明らかにならなければ, サービスを追加する際の情報としては不十分である.

4.2 測定環境

測定は以下の計算機上で行った.

Hand-build PC		
CPU		Intel Celeron Proceccor 300A [16]
	Clock	300MHz
	L1 Cache	32KB
	L2 Cache	128KB
	System Bus Clock	66MHz
Hard Disk		QUANTUM FIREBALL ST3.2A[17]
	Capacity	3.2GB
	Seek Time	10ms
	Rotational Speed	5400rpm
	Interface	Ultra ATA
Main Memory		128MB
OS		FreeBSD[5]2.2.1 RT-Mach[6]MKNG008

FreeBSD を選択した理由として, これが広く一般に利用されている為, 高度に最適化された実用的な OS であると推測できることが挙げられる.

RT-Mach を選択した理由として, これが FreeBSD とバイナリ互換であることが挙げられる. このことは, RT-Mach が FreeBSD と同様の機能を持つということを意味する. その上で, FreeBSD と RT-Mach では特に IPC において, その差が大きく出ることが予測された. FreeBSD はカーネル呼び出しを中心としたサービスの提供方式であるのに対し, RT-Mach では, メッセージ通信を中心としたサービスを提供している. その為, IPC の性能がシステム全体の性能に大きく影響する. もし, IPC の速度が遅ければ, その速度は大幅に遅くなるはずである. しかし, 体感速度はさほどでもない. このことから, RT-Mach の IPC が最適化されているであろうと推測した.

4.3 測定項目

測定では, 単に整数を返し実際の処理は何も行わない関数とシリアルポートのレジスタを読み再び書き戻す関数をカーネル, サーバ, ライブラリ内で実装した.

測定プログラムの骨組み

図 4.1 に測定プログラムの簡単な骨格を示す. 測定値の取得には, プロセッサに内蔵された動作周波数と同じ精度を持つタイムスタンプカウンタ [18] を用いた. 測定に利用したカウンタのマクロを付録 A に示す. 尚, 以下に示すデータは特別の指定がない限り, タイムスタンプカウンタの値をマイクロ秒で表したものである. 測定に利用した CPU は, FreeBSD の起動時に 300.688565MHz と言う結果を出している. 従って, 実測値は値に 300.688565 をかけたものと言う事になる.

シリアルポートへの読み書き

実際の周辺機器への読み書きの時間費用を測定する為に, 今回は Serial Port のレジスタへの読み書きを選択した. これに用いた関数を図 4.2に示す. シリアルレジスタの読み書きを実現する為に, ユーザアドレス空間からカーネルの管理するアドレス空間をアクセス可能とする /dev/io を利用した.

通常, 利用者空間からカーネル空間にアクセスすることはできない. このデバイスは CPU のフラグを変更することにより, CPU の例外生成を抑制する. FreeBSD のマニュアルの 4 節には次のように記載されている.

> The special file /dev/io is a controlled security hole that allows a process to gain I/O privileges (which are normally reserved for kernel- internal code). Any process that holds a file descriptor on /dev/io open will get its IOPL bits in the flag register set, thus allowing it to perform direct I/O operations. This can be useful in order to write userland programs that handle some hardware directly.

測定値には /dev/io を open する時間は含まれない.

LKM

カーネルサービスを測定するプログラムでは FreeBSD, RT-Mach 両方で LKM を利用した. LKM は, Linux[10] 等では kerneld と共に一般的に利用されている機構である. FreeBSD の LKM では syscall(2) をオフセットを引数に呼び出し, RT-Mach の LKM では lkm_call_generic_fast() を呼び出した.

測定値には, LKM をカーネル空間に置き初期化する時間は含まない.

付録 Bに測定に利用した FreeBSD 上での LKM のソースファイルを示す. 付録 Cに RT-Mach の LKM のソースファイルを示す.

IPC

IPC の測定として FreeBSD では Berkeley Socket で UDP (User Datagram Protocol) を INET domain で利用した. ソケットの初期化や bind(2) の時間は測定値には含まず, sendto(2) と recvfrom(2) が終了した時間を測定値として利用した. 今回の測定では, 異なる実行主体間の安全なデータの転送の時間費用を測定することが目的であり, 到着順序保証や輻輳制御と言った高度な機能は対象外である. 従って, TCP (Transmission Control ProtocoL) ではなく UDP を利用した.

RT-Mach では, port を利用した RPC (Remote Procedure Call) を使った. ソースコードは, サーバ, クライアント, MIG (Mach Interface Generator) 定義ファイルの三つである. MIG は利用者が記述した定義ファイルから, 通信の為に必要なコードを自動的に生成するプログラムである. Mach port 通信には, 同期, 非同期の両方があるが, 今回は, MIG 定義ファイルの中で “routine” を記述し, 同期をとった場合の測定を行った.

FreeBSD, RT-Mach と共にサーバを起動する時間は測定値には含まない.

測定はそれぞれの項目に付いて更に三つの測定をした. 初回呼び出し時間, 初回から 30 回までの呼び出し時間, 4 回目以降の呼び出し時間である.

初回呼び出し時間

初回呼び出し時間として, 図 4.1 中の, 変数 turn を 1 とし, このプログラムを 1000 回起動し, その結果を記録した.

初回から 30 回目までの時間費用

時間費用の時系列的な変化を測定する為に, 初回から 30 回目までの測定値を記録した. この測定では, 図 4.1 中の変数 turn を 30 とし, これを 50 回起動し, 時系列的に値を記録した.

4 回目以降の時間費用

各項目の初期化費用を除いた値を測定する為, 4 回目から 1003 回目までの時間費用の測定をした. この測定では, 図 4.1 中の変数 turn を 1003 とし, これをそれぞれの項目毎に一回起動し, その結果を記録した. 図 4.3 より, 初回から 3 回目までは, 以降の結果との大きな違いから初期化費用が関係していると判断し結果から取り除いた.

```
#include <stdio.h>
#include "pencnt.h"

extern int call(void);

int main(argc, argv)
int argc;
char **argv;
{
    int i,ret,turn;

    turn=atoi(argv[2]);

    initilize();

    for(i=0;i<turn;i++){
        init_pen_count();
        ret = call();
        printf("%lu ",get_pen_count());
    }

    finalize();

    return ret;
}
```

図 4.1: 測定プログラムの枠組

```
static __inline unsigned char inb(unsigned int port);
static __inline void outb(unsigned int port, unsigned data);

#define BASE 0x2f8

int read_sio_reg(unsigned char data[8]){
    int i;
    for(i=0;i<8;i++) {
        data[i]=(unsigned char)inb(BASE+i);
    }
    return(0);
}

int write_sio_reg(unsigned char data[8]){
    int i;
    for(i=0;i<8;i++) {
        outb(BASE+i, data[i]);
    }
    return(0);
}

static __inline  unsigned char
inb(unsigned int port)
{
    unsigned char data;
    __asm __volatile("inb %%dx,%0" : "=a" (data) : "d" (port));
    return (data);
}

static __inline void
outb(unsigned int port, unsigned data)
{
    unsigned char al;
    al = data;
    __asm __volatile("outb %0,%%dx" : : "a" (al), "d" (port));
}
```

図 4.2: シリアルレジスタへの読み書きの手続き

4.4 測定結果

表 4.1 に, 初回呼び出しにかかる時間費用を示す. 表の値は 1000 回の平均, 最大, 最小, 標準偏差である.

表 4.1: 測定結果 初回時間費用 (usec)

	bsd-null-stlib	bsd-null-shlib	bsd-null-lkm	bsd-null-ipc
AVG	0.266	4.764	13.852	158.484
MAX	0.269	5.364	37.258	281.028
MIN	0.233	4.446	12.684	145.506
STD	0.003	0.166	1.198	8.067
	mach-null-stlib	mach-null-shlib	mach-null-lkm	mach-null-ipc
AVG	0.272	6.121	3.692	30.106
MAX	20.303	102.082	21.577	137.212
MIN	0.233	4.959	3.133	27.853
STD	0.854	3.188	0.968	5.616

	bsd-r-stlib	bsd-r-shlib	bsd-r-lkm	bsd-r-ipc
AVG	14.645	18.72	36.655	176.468
MAX	73.737	52.799	65.935	227.9
MIN	13.858	17.896	34.418	164.403
STD	2.637	1.308	1.674	6.167
	bsd-w-stlib	bsd-w-shlib	bsd-w-lkm	bsd-w-ipc
AVG	14.567	18.816	36.705	177.479
MAX	47.514	51.239	70.395	326.231
MIN	13.842	17.882	34.617	160.209
STD	1.508	2.296	2.81	10.168

表 4.1 は, 初回呼び出し時間 1000 回の平均, 最大, 最小, 標準偏差を表す. 表中の表記は, AVG: 平均, MAX: 最大値, MIN: 最小値, STD: 標準偏差, bsd:FreeBSD, mach:RT-Mach, stlib: 静的ライブラリ, shlib: 共有ライブラリ, lkm:LKM, ipc:IPC, null: 空の呼び出し, r: シリアルレジスタからの読み込み, w: シリアルレジスタへの書き込み である. 又, 値の単位はマイクロ秒である.

次に初回から 10 回までの時系列的な時間費用を図 4.3 に示す. x 軸は回数, y 軸は時間費用を表す.

4 回目からの 1003 回目までの測定値を表 4.2 に示す. 表の値は 1000 回の平均, 最大, 最小, 標準偏差である.

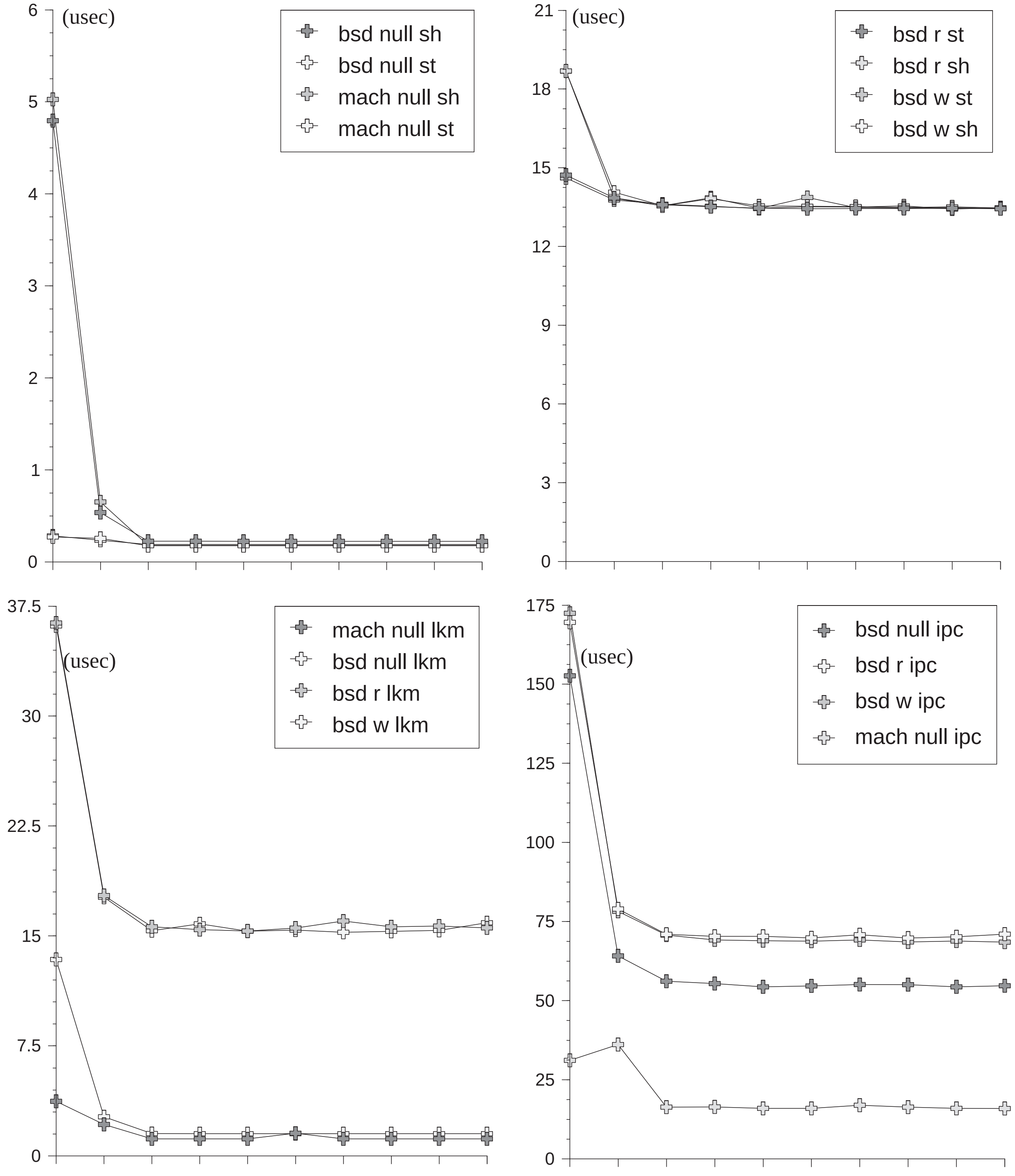

図 4.3: 平均値の初回から 10 回まで

図4.3 は, 初回呼び出しから 10 回目までそれぞれ 50 回測定した測定値の平均を表す． 表中の表記は, bsd:FreeBSD, mach:RT-Mach, stlib: 静的ライブラリ, shlib: 共有ライブラリ, lkm:LKM, ipc:IPC, null: 空の呼び出し, r: シリアルレジスタからの読み込み, w: シリアルレジスタへの書き込み である． 又, y 軸の単位はマイクロ秒である．

表4.2: 測定結果4回から1000回目までの時間費用 (usec)

	bsd-null-stlib	bsd-null-shlib	bsd-null-lkm	bsd-null-ipc
AVG	0.176	0.224	1.513	52.957
MAX	0.176	0.226	24.327	85.56
MIN	0.176	0.179	1.49	51.678
STD	0	0.009	0.722	2.501
	mach-null-stlib	mach-null-shlib	mach-null-lkm	mach-null-ipc
AVG	0.189	0.180	1.165	15.714
MAX	0.189	0.180	7.965	92.451
MIN	0.189	0.180	1.157	15.355
STD	0	0	0.216	2.556

	bsd-r-stlib	bsd-r-shlib	bsd-r-lkm	bsd-r-ipc
AVG	13.704	13.588	15.851	67.103
MAX	190.366	47.305	199.309	227.056
MIN	13.203	13.196	15.099	65.852
STD	5.62	1.255	5.833	5.317
	bsd-w-stlib	bsd-w-shlib	bsd-w-lkm	bsd-w-ipc
AVG	13.57	13.737	15.419	67.817
MAX	44.112	170.502	38.355	229.364
MIN	13.196	13.196	14.899	65.809
STD	1.292	1.255	0.981	5.428

表4.2は,4回目から1003回目までの1000回の平均, 最大, 最小, 標準偏差を表す. 表中の表記は, AVG: 平均, MAX: 最大値, MIN: 最小値, STD: 標準偏差, bsd:FreeBSD, mach:RT-Mach, stlib: 静的ライブラリ, shlib: 共有ライブラリ, lkm:LKM, ipc:IPC, null: 空の呼び出し, r: シリアルレジスタからの読み込み, w: シリアルレジスタへの書き込み である. 又, 値の単位はマイクロ秒である.

4.5 考察

測定結果の考察は次のような五項目について行った. 本節では各項目の考察詳細を述べる.

- 初回呼び出しと 4 回目以降の呼び出しの比.
- 呼び出しに掛かる費用と仕事量の割合.
- 処理時間の差の総和.
- IPC と LKM の比較.
- 外れ値の頻度.

4.5.1 初回呼び出しにかかる割合

図 4.4 に “初回平均時間費用” ÷ “4 ~ 1000 回平均時間費用” を示した. 初回呼び出し時平均時間費用を C_1, 4 ~ 1000 回平均時間費用を C_n とすると, 上の x 軸は C_1,C_n , 下の x 軸は

$$x_1 = \frac{C_1}{C_n} \tag{4.1}$$

に対応する. これを見ると共有ライブラリの初回呼び出し時間費用が, その 4 回目以降の平均と比べて, 特に多く掛かっている. RT-Mach では初回呼び出し時間だけの比較では共有ライブラリよりも LKM の呼び出しの方が高速である.

図 4.3を見ると, 共有ライブラリで時間費用が安定するのは 3 回目以降である. 従って, 共有ライブラリでサービスを実現する事はその粒度と頻度, 回数に充分留意する必要がある. 共有ライブラリの初回呼び出し時間と 4 回目以降の呼び出し時間の割合は他のそれと比べても極端に大きい.

一つの実行主体のみの利用しかなく, 尚且つ, それが利用される回数が 3 回以下である様なサービスは, それが静的ライブラリで実現可能なものならば共有ライブラリで実現するべきではない. しかし, 連続メディアを扱うアプリケーション等, 同じ呼び出しが繰り返し利用され, しかも, 初回呼び出しよりもその後の安定の方が重要視されるサービスでは, 上の例には当てはまらない.

又, ライブラリの大きさ, 或はサービスの粒度にも配慮する必要がある. 幾つかの手続きを一つのライブラリとしてまとめる場合, その数が充分な大きさでないと共有ライブラリの時間的費用は静的ライブラリのそれに劣る.

時間費用のみを見ると, 静的ライブラリは粒度が細かく, 利用の頻度が高く回数が少ないサービス, 共有ライブラリは粒度が荒く, 利用の頻度が低く回数が多いサービスが最適であるといえる. これは, メモリを占有してしまう静的ライブラリ, 共有可能な共有ライブラリと言う定性とも一致する.

初回時間費用から見た共有ライブラリと静的ライブラリに適したサービスの特性.

ライブラリの種類 \ サービスの特性	粒度	頻度	回数
共有ライブラリ	粗	高	多
静的ライブラリ	細	低	少

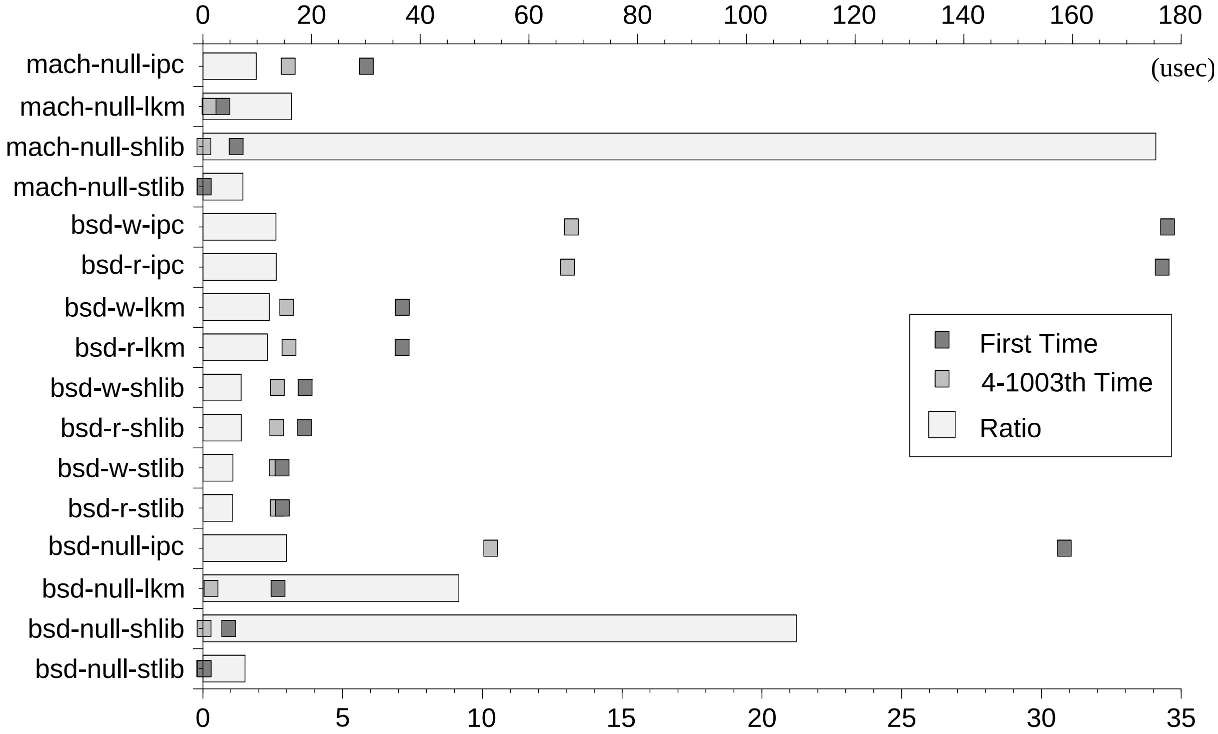

図 4.4: 初回呼び出しと 3-1000 回目の呼び出しとの割合

図 4.4 は “初回平均時間費用” ÷ “4 ~ 1000 回平均時間費用” を示す． 初回呼び出し時平均時間費用を C_1, 4 ~ 1000 回平均時間費用を C_n とすると, 上の x 軸は C_1,C_n , 下の x 軸は $\frac{C_1}{C_n}$ に対応する． 表記は, bsd:FreeBSD, mach:RT-Mach, stlib: 静的ライブラリ, shlib: 共有ライブラリ, lkm:LKM, ipc:IPC, null: 空の呼び出し, r: シリアルレジスタからの読み込み, w: シリアルレジスタへの書き込み である． 又, 上の x 軸の単位はマイクロ秒である．

4.5.2 呼び出しに掛かる費用と仕事量の割合

サービスの総経費は, サービス提供経費 + サービス内容処理の経費と考える事ができる． サービス提供経費とは, サービスを提供する為に最低限必要とされる費用である． これは, サービス提供方式によって異なる． サービス内容処理経費とは, 実際にサービスを処理する為の時間であり, サービス内容により異なる． これは, 提供方式の違いには依存しない． 総経費とサービス提供経費とサービス内容処理経費の関係を式で表すと,

$$\text{総経費} = \text{サービス提供経費} + \text{サービス内容処理経費} \tag{4.2}$$

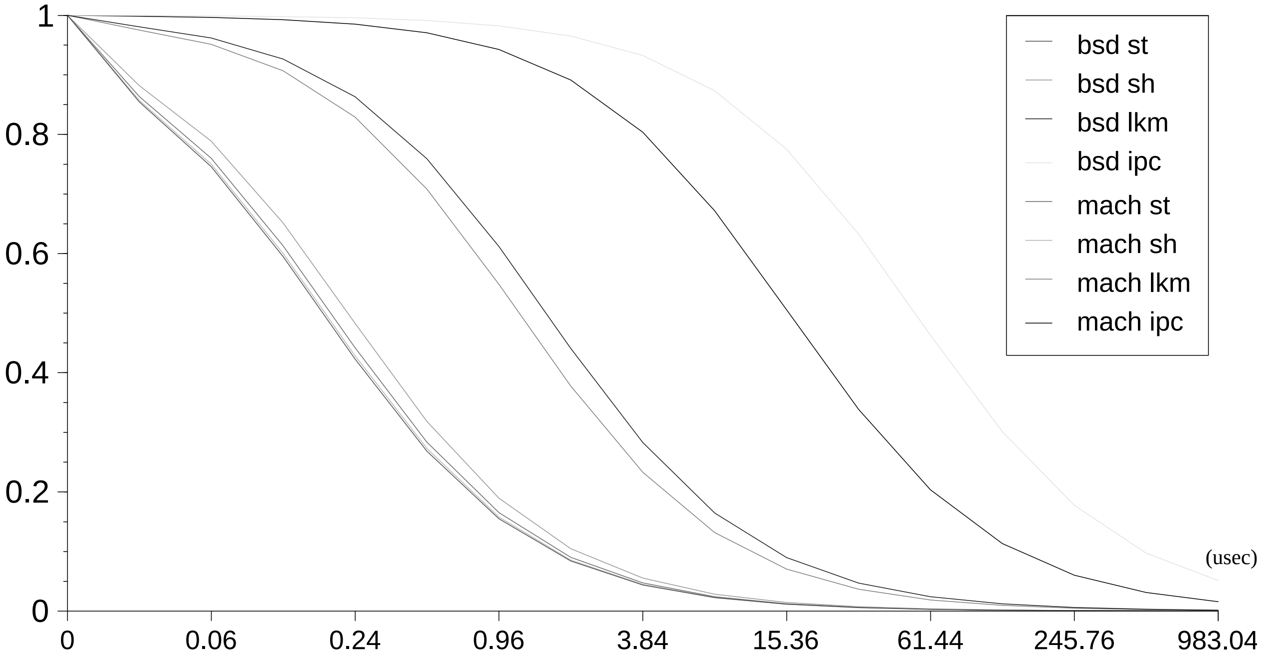

図 4.5: サービス提供経費の総経費に占める割合

図4.5 は, サービス提供経費を O, サービス内容処理経費を T とし, 総経費に占めるサービス提供経費の割合は $\frac{O}{O+T}$ とする時, この値を y 軸に, T の値を x 軸に取って図表に表したものである. 表記は, bsd:FreeBSD, mach:RT-Mach, st: 静的ライブラリ, sh: 共有ライブラリ, lkm:LKM, ipc:IPC である. 又,x 軸の単位はマイクロ秒である.

の様になる.

空の呼び出しをサービス提供の為の経費と考え, サービス提供経費を O, サービス内容処理経費を T とすると総経費に占める提供経費の割合 R は

$$R = \frac{O}{O+T} \tag{4.3}$$

となる. この値を y 軸に, T の値を x 軸に取って図表に表すと図 4.5 の様になる. 例えば, 今回, 測定したシリアルポートのレジスタへの読み書きの 4 倍程度の処理時間, 即ち, T の値が 100usec の場合には, R の値はライブラリサービスでは FreeBSD,RT-Mach と共に 0.2%, LKM では FreeBSD で 1.5%, RT-Mach で 1.2%, IPC では FreeBSD,34.6%, RT-Mach,13.6% となる.

又, T が増加すればするほど, それぞれの方式の総経費の比は 1 に近づく. 静的ライブラリのサービス提供経費を O_{st}, 共有ライブラリのサービス提供経費を O_{sh}, LKM のサービス提供経費を O_l,IPC のサービス提供経費を O_i とすると,

$$O_{st} + T : O_{sh} + T : O_l + T : O_i + T \tag{4.4}$$

は 1:1:1:1 に近付く. これは,T の大きい測定ではその測定値があまり変わらないとする Kaashoek 1997[9] の結果と一致する. これを, 表 4.3 に表した.

従って, 処理時間を多く要するようなサービスでは, 相対的に, 提供方式の持つ高速であると言う利点を失う.

表 4.3: サービス提供経費が変化した時の総経費の比
(FreeBSD / RT-Mach)

	10usec	100usec	1000usec
$\frac{C_{sh}}{C_{st}}$	1.00 / 1.00	1.00 / 1.00	1.00 / 1.00
$\frac{C_{lkm}}{C_{st}}$	1.13 / 1.10	1.01 / 1.01	1.00 / 1.00
$\frac{C_{ipc}}{C_{st}}$	6.19 / 2.52	1.53 / 1.15	1.05 / 1.02
$\frac{C_{lkm}}{C_{sh}}$	1.13 / 1.10	1.01 / 1.01	1.00 / 1.00
$\frac{C_{ipc}}{C_{sh}}$	6.16 / 2.53	1.53 / 1.16	1.05 / 1.02
$\frac{C_{ipc}}{C_{lkm}}$	5.47 / 2.30	1.51 / 1.14	1.05 / 1.01

表4.3 は, 二つのサービスの総時間の比を表す. この値は, サービス x とサービス y の, $\frac{C_x}{C_y}$ である. C は, サービス提供経費 C_x とサービス内容処理経費 T の和である. / を挟んで前が, FreeBSD の値, 後が RT-Mach の値である.

4.5.3 処理時間の差の総和

サービス提供経費とサービス内容処理経費との和であるサービス全体の時間費用の比が小さくなったとしても, 時間費用の差は縮まない. 提供方式 x のサービス提供経費を O_x, x のサービスの総経費を $C_x = O_x + T$ とする. n 回目までの総経費の総和を $\sum C_n$ とすると, x と y の総経費の総和の差は,

$$\sum(C_x - C_y) = (O_x - O_y) \times n \tag{4.5}$$

という単純な計算で求めることができる.

これを表にしたものが, 表4.4 である.

短期間に多数回呼び出されるようなサービスでは, サービス提供経費の小ささが重要な要素となる. しかし, 今回の測定を見る限りに置いて, サービス提供経費の差の総和以外のことに注目するべきである. 例えば, RT-Mach の LKM と共有ライブラリでは100 マイクロ秒の差が付くのに, 100 回を要する. 100 マイクロ秒は, 今回, 測定したシリアルレジスタへの読み書きの 2 倍程度の仕事, 即ち, 30 マイクロ秒程度の仕事をするとして, その 100 回分の時間の 3% 強である.

但し, IPC とその他の提供方式との時間差が大きいことには注意するべきである. 今回の測定では, サーバからの返答を待って, 要求を出している. しかし, 多重サービスを行う為には, サーバが新たなるコンテキストを作り, 作業することも考えられる. この場合には, IPC の応答時間は更に高まることも予測される.

4.5.4 IPC と LKM

Mach 等の IPC を利用したサービスを行う OS では, IPC とシステムコールの処理速度の差が性能に大きく影響する [1]. 又, 実際, RT-Mach と FreeBSD では IPC の速度に差がある.

IPC のサービス提供経費が変化することを考える. IPC のサービス提供経費を O_i, LKM のサービス提供経費を O_l とし, 総経費の比 R は,

$$R = \frac{O_i + T}{O_l + T} \tag{4.6}$$

の様に表すことができる. R は, $O_l = 1.16 \quad O_i = 15.76 \quad T = 15(usec)$ (RT-Mach の場合に近い値) の時, 1.90 となる. 又, $T = 25 \quad T = 40 \ (usec)$ (今回の測定のシリアルへの読み書きの 2 倍,3 倍に近い値) の場合にはそれぞれ, 1.56,1.35 となる. $T = 15(usec)$ とし, $O_i = 10$ (Mach IPC の 60 %程度の速度) の時も同様にして求めると, その値は 1.79 となる.

$O_l = 350, \ x = T, \ y = \frac{O_i+T}{O_l+T}$ とし, $O_i = 10, 15, 50$ の値を取る時, 図表を書くと図 4.6 の様になる. 例えば, IPC が RT-Mach より 60% 高速となり, サービス提供経費が 90 マイクロ秒だとし, LKM の値が RT-Mach と変わらないとすると, LKM と IPC を利用した場合の費用の総和の比では, LKM が 10% 程度の高速性しか持たない.

前述の様に, 提供方式を実現する為のサービス提供経費とサービス内容処理経費との和であるサービス全体の時間費用の比が小さくなったとしても, 時間費用の差は縮まない. 表 4.4 の計算式より, その差が 100 マイクロ秒に達するのは FreeBSD で 1 回目, RT-Mach で 6 回目, 1 ミリ秒に達するのは FreeBSD で 17 回目, RT-Mach で 67 回目である.

サービス提供経費が大きいサービスや, 高速な IPC を備えているシステム, 或はその両方の条件を満たす場合には, LKM と IPC の時間比は縮まる. しかし, 頻繁に利用されるようなサービスを IPC として実装することは, その時間費用の差が重なり, システムの性能は落ちる.

4.5.5 外れ値の頻度

平均を $\bar{C}$ とし, 標準偏差を S とすると, 標本中の値の一つである C_n の検定統計量 R は

$$R = \frac{\bar{C} - C_n}{S} \tag{4.7}$$

として表現する事ができる.

表 4.2 に表現したデータを基に, それぞれのデータから大きいものから 6 つ選択し, 検定統計量 T を求めると表 4.5 の様になった. FreeBSD の静的ライブラリで空の呼び出しを測定したデータと, RT-Mach での空呼び出しを静的, 共有ライブラリを測定したデータ

では, 標準偏差が 0 である為, 計算不可能となった. 又,FreeBSD での共有ライブラリで空呼び出しを測定したデータでは, 測定結果が 68 と 54 の二つに離れて, 更にこの二つのみの測定値しか出なかった為, T の値が 1.99 とかなり大きく, 尚且つ上から 6 番目までの T の値は全て 1.99 となった. ライブラリを用いた空呼び出しの頻度に付いては図 4.7 に示す. 図 4.7 中の x 軸は測定値を, y 軸はその測定値が何回記録されたかを示す. 図中で一番濃い色の棒が FreeBSD で共有ライブラリを利用した空呼び出しである.

上記の二つ,FreeBSD 上でのライブラリを用いた空の呼び出し以外では, IPC の T の値が他のそれを上回っている. 特に, 他の値が, 上から 5 番目以降はいずれも 1 を割り込むのに対し, IPC ではいずれもその値の落ち込みは緩やかである. これは IPC がばらついた値を取りやすい事を意味している.

従って, ばらついた値の出やすい IPC では, 例えば実時間性を求められる処理等, 安定性を求められる処理には向かない.

表4.4: 二つのサービスの時間差が1ミリ又は100マイクロ秒を越える回数

FreeBSD 1msec

	Static Library	Shared Library	LKM	IPC
Static Library	\	20684	739	16
Shared Library	-20684	\	770	17
LKM	-739	-770	\	17
IPC	-16	-17	-17	\

RT-Mach 1msec

	Static Library	Shared Library	LKM	IPC
Static Library	\	-99644	1023	63
Shared Library	99644	\	1013	63
LKM	-1023	-1013	\	67
IPC	-63	-63	-67	\

FreeBSD 100usec

	Static Library	Shared Library	LKM	IPC
Static Library	\	1985	65	-1
Shared Library	-1985	\	71	-1
LKM	-65	-71	\	-1
IPC	1	1	1	\

RT-Mach 100usec

	Static Library	Shared Library	LKM	IPC
Static Library	\	-9437	100	5
Shared Library	9437	\	100	5
LKM	-100	-100	\	6
IPC	-5	-5	-6	\

表4.4は, 二つのサービスの時間差が1ミリ又は100マイクロ秒を越える時を表す. この値は, サービスxのn回目の呼び出し時間をx_nとし, 平均呼び出し時間を$\bar{x}$, 時間差をdとする時, $\frac{d-|x_1-y_1|}{\bar{x}-\bar{y}}+1$で求められる値を, 小数点以下切り上げたものである.

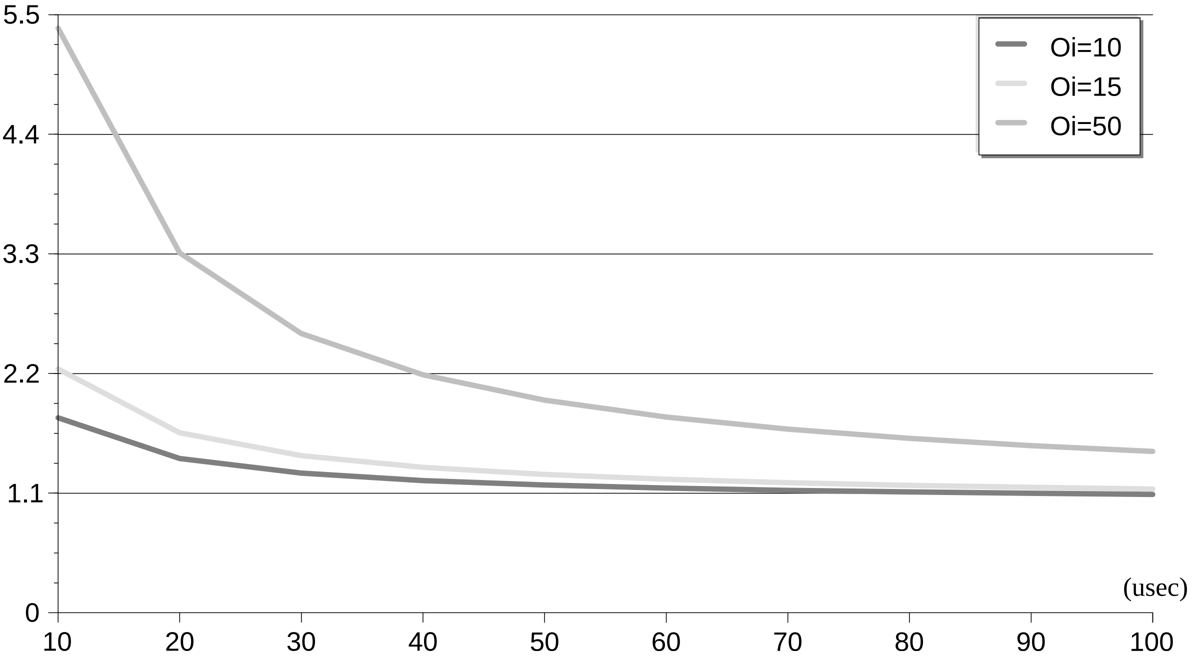

図 4.6: IPC と LKM の処理の割合

図 4.6 は $O_l = 350,\ x = T,\ y = \frac{O_i+T}{O_l+T}$ とし, $O_i = 10, 15, 50$ の値を取る時のものである． O_l は, LKM のサービス提供経費, O_i は IPC のサービス提供経費, T はサービス内容処理経費とする.

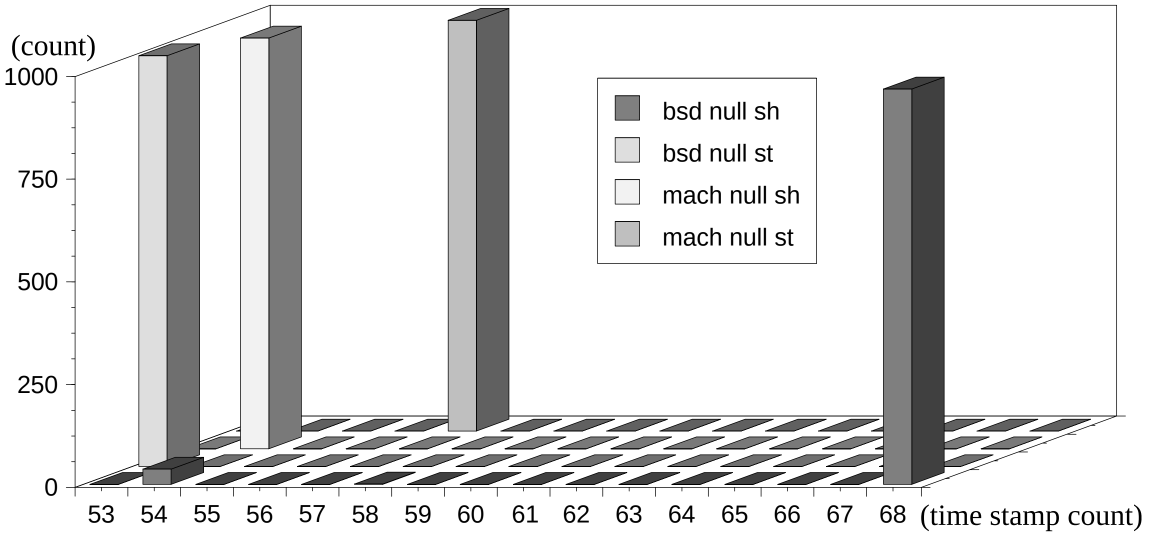

図 4.7: ライブラリを用いたから呼び出しの頻度

図 4.7 は, ライブラリを用いた空呼び出しの頻度に付いて示したものである．図中の x 軸は測定値を, y 軸はその測定値が何回記録されたかを示す．尚, x 軸の単位は簡略化の為に実測値, 即ちタイムスタンプカウンタの値を利用している.

表 4.5: 各項目毎の値のばらつき

	bsd-null-stlib	bsd-null-shlib	bsd-null-lkm	bsd-null-ipc
	ERROR	0.199	-0.032	5.994
	ERROR	0.199	-0.032	7.015
	ERROR	0.199	-0.032	8.215
	ERROR	0.199	-0.032	9.906
	ERROR	0.199	-0.032	12.607
	ERROR	0.199	0.014	13.011
MAX	ERROR	0.199	31.591	13.036
	bsd-r-stlib	bsd-r-shlib	bsd-r-lkm	bsd-r-ipc
	0.097	0.533	0.054	2.209
	0.097	0.536	0.054	2.21
	0.097	0.536	0.054	2.304
	0.097	6.396	0.068	2.468
	1.666	8.905	1.309	3.291
	2.217	10.432	2.126	4.428
MAX	31.434	26.868	31.453	30.086
	bsd-w-stlib	bsd-w-shlib	bsd-w-lkm	bsd-w-ipc
	0.388	0.104	0.325	2.31
	0.388	0.104	0.369	2.582
	0.388	1.64	0.538	2.615
	6.061	1.777	7.879	2.685
	6.499	2.499	8.903	3.456
	17.941	4.481	13.879	4.069
MAX	23.63	31.021	23.394	29.764
	mach-null-stlib	mach-null-shlib	mach-null-lkm	mach-null-ipc
	ERROR	ERROR	-0.034	2.332
	ERROR	ERROR	0.028	2.353
	ERROR	ERROR	0.12	2.647
	ERROR	ERROR	0.244	3.004
	ERROR	ERROR	0.244	3.238
	ERROR	ERROR	1.632	3.537
MAX	ERROR	ERROR	31.544	30.018

表 4.5 は, 各測定項目の上位 6 つの測定値を抽出し, 測定値から項目の平均を引き, それを標準偏差で割った値である. 測定値は表 4.2 のものを利用した. bsd:FreeBSD, mach:RT-Mach, stlib: 静的ライブラリ, shlib: 共有ライブラリ, lkm:LKM, ipc:IPC, null: 空の呼び出し, r: シリアルレジスタからの読み込み, w: シリアルレジスタへの書き込み である. FreeBSD での空呼び出しを静的ライブラリ, RT-Mach での空呼び出しを静的, 共有ライブラリで測定した値は, 標準偏差が 0 となった為, エラーとなった.

第 5 章

結論

本章では, 本論文の結論を述べる. 又, 今後の課題について述べる.

5.1 まとめ

単層カーネルの問題点は, 動的的適応性, 拡張性及び特化性の欠如している点にある. その為, 様々な方式が提案されている. しかし, 問題は個々の提供方式の是非ではなく, サービスの特性に合致した提供方式の選択方法である. 本研究の目的は, 各サービス提供方式の特徴を明確にし, サービス特性と提供方式の関係を明らかにすることであった.

提供方式の特徴を明らかにする為には, まず, 提供方式を分類し, その詳細について明確にする必要があった. 本研究では, サービス提供方式をアドレス空間により分類し, 既存のサービス提供方式を参考にしながら定性的評価を行い, 定量的評価として時間費用の測定を行い, その分析をした.

5.2 測定結果と定性的評価のまとめ

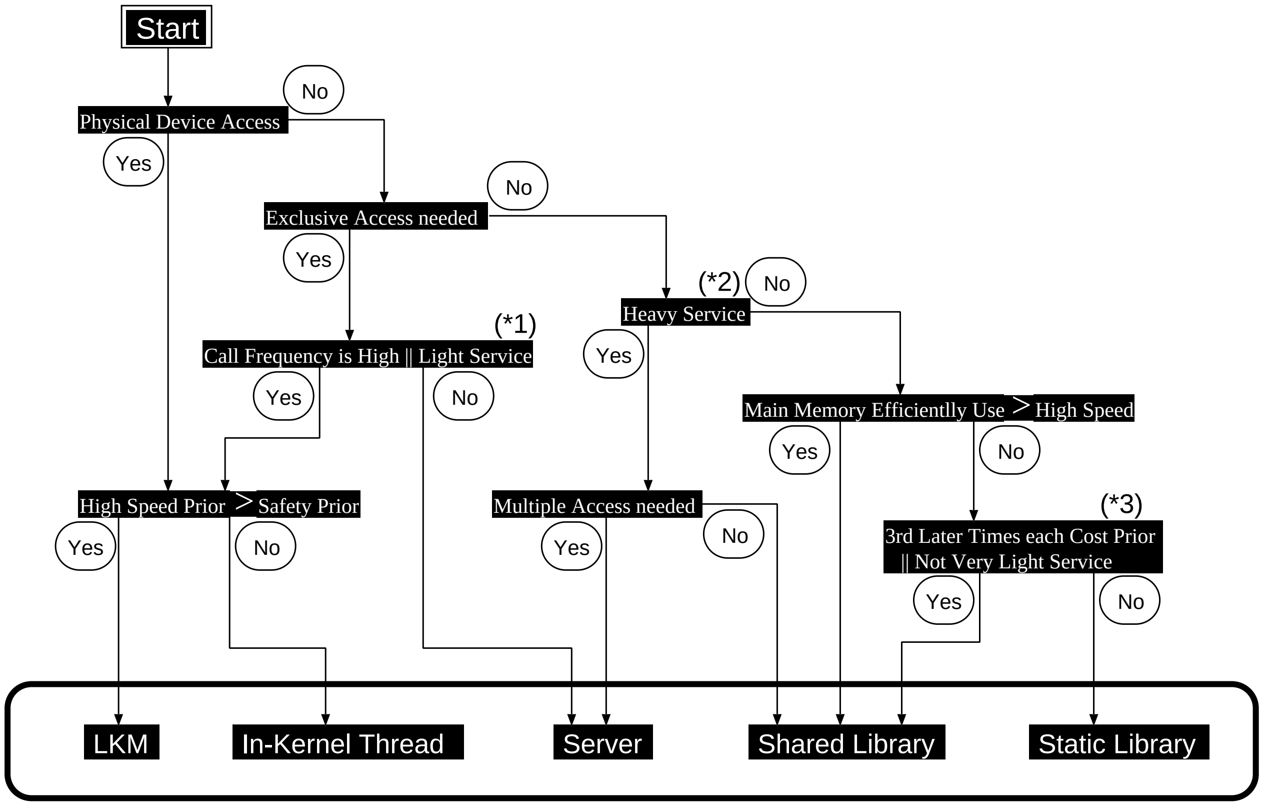

図 5.1: 提供方式選択チャート

図 5.1は, サービス配置方針を簡単な図として表現したものである．利用者は, Start から始め, 追加したいサービスの特徴に関する簡単な質問に Yes No で答えることで, そのサービスに最適な提供方式を知ることができる．

(*1) IPC を利用した場合の全時間費用を T_{ipc}, LKM を利用した場合の全時間費用を T_{lkm}, 純粋な処理の為の費用を C_{body}, 個々の方式を利用する為の費用をそれぞれ C_{ipc},C_{lkm} とすると, $T_{lkm} = C_{lkm} + C_{body}$, $T_{ipc} = C_{ipc} + C_{body}$ となる．$\frac{T_{ipc}}{T_{lkm}}$ が 1.1 となるのは, FreeBSD で $C_{body} = 513$ マイクロ秒, RT-Mach で $C_{body} = 144$ マイクロ秒である．

IPC と LKM で時間費用の n 回目までの総和の差, $\sum(T_{ipc} - T_{lkm})$ は, $(C_{ipc} - C_{lkm}) \times n$ と考えることができる．表 4.4 によれば, この値が 1 ミリ秒になるのは, FreeBSD で 17 回目, RT-Mach で 67 回目である

(*2) IPC を利用した場合の, 全時間費用を T_{ipc}, 共有ライブラリを利用した場合の, 全時間費用を T_{shlib}, 純粋な処理の為の費用を C_{body}, 個々の方式を利用する為の費用をそれぞれ C_{ipc},C_{shlib} とすると, $T_{shlib} = C_{shlib} + C_{body}$, $T_{ipc} = C_{ipc} + C_{body}$ とな

る. $\frac{T_{ipc}}{T_{shlib}}$ が 1.1 を割るのは, FreeBSD で $C_{body} = 528$ マイクロ秒, RT-Mach で $C_{body} = 156$ マイクロ秒である. IPC と共有ライブラリで時間費用の n 回目までの総和の差, $\sum(T_{ipc} - T_{shlib})$ は, $(C_{ipc} - C_{shlib}) \times n$ と考えることができる. 表 4.4この値が 1 ミリ秒になるのは, FreeBSD で 19 回目, RT-Mach で 64 回目である

(*3) 共有ライブラリに比べて静的ライブラリが優位な条件は, 次の通りである. 呼び出し回数が少なく, 更にサービスは極端に小さい場合である.

例えば, 静的ライブラリの必要経費を O_{st}, 共有ライブラリの必要経費を O_{sh}, 純粋なサービス実現に掛かる費用を T, 一回の呼び出しに掛かる時間を $O + T$ とする. O_{sh} O_{st} を図 4.3から求め, $T = 15$ とし, 10 回呼び出されたとすると, $\frac{\sum(O_{sh}+T)}{\sum(O_{st}+T)}$ を求めると, FreeBSD では 1.034, RT-Mach では 1.033 である.

5.3 今後の課題

本研究では, 各提供方式の純粋な上乗せ費用を測定する為に, 空の呼び出しとシリアルレジスタへの書き込みについて測定した. その為, 実際の OS がどの程度の付加の掛かるサービスを提供しているのか, 或は理論値は正しいのかに付いては今後の課題となる.

どの程度の粒度でサービスを提供すれば良いのかも今後の課題として残った. 例えば, 新たなるファイルシステムの実装を考えた場合, 本研究では, それが抽象化と物理機器制御に適切に分離されていることが前提となる. この分離が適切なのか, 或は更に粒度の細かい分離が必要なのかについても, 今後の研究課題である.

第 6 章

謝辞

本研究を進めるにあたり、御指導をいただきました、慶應義塾大学環境情報学部教授の徳田英幸博士と村井純博士に感謝いたします。また、絶えず御助言と御指導をいただきました、同学部の中村修博士と楠本博之博士に深い感謝の意を表します。

杉浦一徳氏, 小川晃通氏をはじめとする慶應義塾大学環境情報学部の徳田・村井・中村・楠本研究室の UOS Group の方々, 望月祐洋氏, 大越匡氏, 中沢仁氏をはじめとする同研究室 KMSF Group の方々, 並びに同研究室のその他の方々にも、本研究を進めていく上で様々な議論をして頂き、また耐えざる励ましとご援助を頂きました。ここに、深い感謝の念を表します。

ありがとうございました。

参考文献

[1] David Golub, Randall Dean, Alessandro Forin, and Richard Rashid. Unix as an application program. In *Proceedings of the USENIX Summer Conference*, June 1990.

[2] D.R Engler and M F. Kaasheoek. Exterminate all operating system abstractions. In *Proceedings of the 5th IEEE Workshop on Hot Topics in Operating Systems*, 1995.

[3] 追川修一, 杉浦一徳, 西尾信彦, 徳田英幸. マイクロカーネルにおけるカーネルモジュールのサポート. 第 53 回情報処理全国大会論文集, 1996.

[4] J. Lepreau, M. Hilber, B. Ford, and J. Law. In-kernel servers on mach 3.0: Implementation and performance. In *Proceedings of the 3rd USENIX Workshop on Micro-kernels and Other Kernel Architectures*, 1993. Santa Fe.

[5] *FreeBSD inc.* http://www.freebsd.org/.

[6] H. Tokuda, T. Nakajima, and P. Rao. Real-time mach: Towards predictable real-time system. In *Proceedings of the USENIX 1990 Mach Workshop*, 1990.

[7] 中島達夫. Real-time mach の現状と今後の課題. In *JAIST IS-RR-94-18S*, 1994.

[8] *IEEE Std 1394-1995, High Performance Serial Bus*, August 1996.

[9] M.F. Kaasheoek, D.R. Engler, and R.G. Gregory. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th Symposium on Operating Systems Principles*, 1997.

[10] *Linux.* http://www.linux.org/.

[11] B.N. Bershad, S. Savage, P. Pardyak, and other. Extensibility, safety and performance in the spin operating system. In *Proceedings of the fifteenth ACM Symposium on Operating Systems Principles*, pp. 267–284, December 1995.

[12] 黒岩, 石井, 高野, 横田. Real-time mach におけるカーネル内タスクの実装と評価. 情報処理学会コンピュータシステムシンポジウム, pp. pp.43–50, 1997.

[13] Y. Yokote. The apertos reflective operating system: The concept and its implementation. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, 1992*, 1992.

[14] Chris Maeda and Brian N. Bershad. Protocol service decomposition for high-performance networking. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pp. 244–255, December 1993.

[15] 伊藤修一, 光澤敦. 可変ページを用いた仮想記憶管理に関する研究. 第57回情報処理全国大会論文集, pp. 1–97, 1998.

[16] *Intel Celeron Processer 300A*. ftp://download.intel.com/design/celeron/datashts/24365806.pdf.

[17] *QUANTUM FIREBALL ST3.2A*. http://www.quantum.com/products/hdd/fireball_st/.

[18] Intel Corp. *Pentium Processor Family Developer's Manual, Volume3: Architecure and Programming Manual*, 1995.

付録 A

測定に利用したマクロ

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/fcntl.h>

unsigned long long start_count, cur_count;

__inline unsigned long long get_pen_count();
__inline unsigned long long get_pen_current();
__inline void init_pen_count();

#define RDTSC(x) __asm __volatile(".byte 0x0f,0x31" : "=A" (x))

inline void
init_pen_count()
{
  RDTSC(start_count);
}

inline unsigned long long
get_pen_count()
{
  RDTSC(cur_count);
  return  cur_count - start_count;
}

inline unsigned long long
get_pen_current()
{
  RDTSC(cur_count);
  return cur_count;
}
```

付録 B

FreeBSD LKM Source File

```
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/exec.h>
#include <sys/sysent.h>
#include <sys/lkm.h>

extern int      mycall __P((struct proc *p, void *uap, int retval[]));
static int      newsyscall_load __P((struct lkm_table *lkmtp, int cmd));
extern int      newsyscall_mod __P((struct lkm_table *lkmtp, int cmd, int ver));

static struct sysent newent = { 0,      mycall };

MOD_SYSCALL( "newsyscall_mod", -1, &newent);

static int
newsyscall_load( lkmtp, cmd)
struct lkm_table        *lkmtp;
int                     cmd;
{
        if( cmd == LKM_E_LOAD){printf( "\nTest null system call\n");}

        return( 0);
}

int
newsyscall_mod( lkmtp, cmd, ver)
struct lkm_table        *lkmtp;
int                     cmd;
int                     ver;
{
        DISPATCH(lkmtp, cmd, ver, newsyscall_load, lkm_nullcmd, lkm_nullcmd)
}
int
mycall( p, uap, retval)
struct proc     *p;
void            *uap;
int             retval[];
{
        return( 0);     /* success (or error code from errno.h)*/
}
```

付録 C

RT-Mach DKM Source File

```
#include <lkm/lkm.h>
#include <mach/kern_return.h>

lkm_trap_t*
simple_entry()
{
        extern lkm_trap_t simple_mothod_table[];
        return simple_mothod_table;
}

static
simple_null()
{
        return 0;
}

static
simple_attach()
{
}

static
simple_detach()
{
}

lkm_trap_t simple_mothod_table[] = {
/* system methods */
        {simple_attach,         0},     /* 0 */
        {simple_detach,         1},     /* 1 */
        {simple_null,           0},     /* 2 */
/* user methods */
        {simple_null,           0},     /* 3 */
};
```