第 13 部

オペレーティングシステム

第 1 章

WIDE におけるオペレーティングシステムの研究

ネットワークに関する研究を続けていく上で、オペレーティングシステムとの関わりを忘れることはできない。Ethernet などのネットワークインタフェースのドライバに始まり、多くのネットワーク機能はオペレーティングシステムの内部で実現される。また、アプリケーションを開発する上でも、オペレーティングシステムが提供するシステムインタフェースが非常に重要な要素である。我々の要求が現存するオペレーティングシステムによってサポートされない場合、その開発自体に密接に関わる必要が生じてくる可能性も少なくはない。そのためには我々が利用しているオペレーティングシステムだけでなく、現在進められている数多くの研究を観察し、必要に応じて積極的に協力していく体制を整えておかなければならない。

1.1 開発環境としてのオペレーティングシステム

ワークステーションが登場する以前、Unix は常にソースと共にあるオペレーティングシステムだった。ユーザは、自分で使用しているシステムに関するすべてのソースプログラムにアクセスすることができた。このことは、Unix が今日これほどまでに多くのユーザに受け入れられるオペレーティングシステムに成長するための原動力であったことに間違いない。Unix の開発者はそれを利用するユーザであり、多くのユーザはある意味で開発者でもあった。ソースコードがあるという利点を生かして、多くの組織が Unix をベースとして実験的な研究を行なってきた。中には派生して別のオペレーティングシステムへと分化していったものもあるし、Unix の標準的な機能として取り入れられたものもある。

一部の研究者の中で利用されていた Unix が、計算機科学を専門としない一般ユーザにまで広く行き渡るようになった大きな原因の一つは、ワークステーションの登場である。高性能低価格のワークステーションの登場により、それまで空調の整った計算機室で管理される比較的高価なスーパーミニコンピュータクラスのマシンで一部のユーザによってだけ使われてきた Unix は、我々の机の上にさえのるようになり、他のユーザの負荷を気にすることなくごく気軽に利用できる存在になったのである。しかし、多くのワークステーションは、バイナリのみでオペレーティングシステムやコマンドが提供された。これによって、今までとは全くことなるユーザ環境を持つ Unix ができあがった。ワーク

ステーション世代の Unix ユーザにとって、ソースコードがないのがあたりまえであり、オペレーティングシステムやシステムコマンドを自分なりにカスタマイズして使用するなどという使い方は思いもよらない。皮肉にも Unix の普及をもたらしたワークステーションによって、Unix 本来の環境が損なわれつつある。

このような状況を解決するために、オペレーティングシステムに関する研究開発を行なっている組織と積極的に協力して、我々が本来の研究を続けていくための開発環境の整備を進めていく。この中には、利用している計算機への移植作業だけではなく、我々が持っている技術をフィードバックするという作業も含まれる。

1.2 開発

上で述べたような開発環境の上で、ネットワーク技術に関する基礎的な研究開発を行なっている。オペレーティングシステム自体を開発する作業は行なっていないが、昨年度は Unix に X.25 のインタフェースを実装し、その上に IP プロトコルを運用するという研究を行なった。今後は新しいデータリンクプロトコルの実装、機密性の高い通信路の確保、ISDN などの新しい通信技術に関する実験などを始めとして、様々な研究分野で、オペレーティングシステムの内部に関係する作業を続けていく。

1.3 調査

現在世界中で数多くのオペレーティングシステムに関する研究が行なわれている。Mach, BSD, V, Amoeba, Sprite などの海外のプロジェクトだけでなく、国内でも ToM, Muse などを始めとして多くの研究が進行している。しかも、これらの中でネットワークに関係しないものは一つとしてない。これからネットワークソフトウェアおよびその運用技術に関する研究を続けていく上で、これらのオペレーティングシステムに関する調査を行なっていく必要があることは明白である。また、要求に応じて我々がもつ技術や考えが、他の研究活動に反映されるように努力していくことも重要である。

第 2 章

Berkeley UNIX

2.1 次期 BSD Unix に関する研究

2.1.1 目的

現在 WIDE プロジェクトではカリフォルニア大学バークレー校の Computer Systems Research Group (CSRG) との共同研究として次期 BSD システムの開発を行なっている。この作業の目的は次のようなものである。

国産ワークステーションのプラットフォーム: 近年のワークステーションの普及により Unix が身近になってきたのは喜ぶべきことであるが、ほとんどのユーザはソースコードなしで Unix を利用している。これは Unix の原則に外れるものであり、ソースコードがあることによって発展してきた Unix の利用形態を崩すという結果になってきている。また各社が別々のプラットフォームを基にして、独自の拡張機能を加えているため、統一性の欠如につながってきている。CSRG では、様々な大学や企業などと協力して、数多くのプラットフォームを次期 BSD システムに取り入れる努力を続けている。これに協力する意味で日本の国産ワークステーションに BSD システムを移植する作業を WIDE で行なっている。

基本ソフトウェア開発のベース: ソースコードのない Unix を使用することは、オペレーティングシステムやネットワークなどの基本ソフトウェアの開発を行なう上では大きな障害となる。WIDE プロジェクトの研究分野には、データリンク、セキュリティなどを始めとしてオペレーティングシステムの内部に及ぶものが少なくない。このため、開発のベースとなるべきオペレーティングシステムが是非とも必要とされている。

X.25: 次期 BSD システムでは OSI ネットワークのサポートとともに X.25 インタフェースに関するソフトウェアが含まれる予定である。一方、WIDE プロジェクトでは以前より学術情報センターネットワークの X.25 網を使った接続に関する研究を行なってきた。X.25 網の利用に関しては日本はアメリカよりも進んでおり、我々が持っている技術を BSD に反映することは非常に意味の深いことである。

2.1.2 作業内容

90 年度は 1990 年 7 月にリリースされた 4.3BSD-Reno を移植するという作業を行っており、現在も続行中である。現在は MIPS の R3000 をプロセッサに持つ SONY の RISC NEWS ワークステーションでを移植の対象としている。

開発マシンには NWS-3860、ターゲットマシンには NWS-3840 を利用している。どちらもメインプロセッサに MIPS R3000、I/O プロセッサに Motorola MC68030 を使用している。コーディング及びコンパイルは NEWS OS 3.9R 上で NEWS OS のコンパイラを用いて行なった。現在はターゲットマシン上でのコンパイルも可能な状態になっているが、コンパイラ自体は NEWS OS のものを利用する。

X.25 に関しては、アメリカの X.25 ネットワークの信頼性が低いということもあって、今年度の日本で行なう開発及び試験は 4.4BSD リリースのために非常に大きな意味を持っている。

2.2 4.3BSD-Reno の概要

4.3BSD-Reno は 1990 年 7 月にリリースされた最新の Berkeley Unix である [110][111]。それまでの最新版は 4.3BSD-Tahoe と呼ばれ、4.3BSD のネットワークコードを新しくしたものだった。4.3BSD-Reno も同様に 4.3BSD に新しい機能を追加したものであるが、表面的な仕様はほとんど変わることのなかった Tahoe 版とは違い、かなり大規模な変更が施されている。

4.3BSD-Reno のリリースは、4.4BSD のための準備と考えられる。4.4BSD では、いくつかの大きな機能が新たにサポートされるが、そのうちの一部を実現したものが 4.3BSD-Reno である。大きな機能としては、ISO/OSI ネットワーキング、ネットワークファイルシステム、POSIX.1 インタフェースのサポートがあげられるが、これら以外にも数多くの変更がなされている。

サポートするハードウェアは以下の通り。

- VAX シリーズ (86x0, 78x, 750, 730; MicroVAX II, 3200/3500/3600; 82x0)
- Tahoe シリーズ (CCI Power 6/32, 6/32SX; Unisys 7000/xx; Harris HCX7, HCX9)
- HP 9000/300 シリーズ

2.2.1 ファイルシステム構造

4.3BSD-Reno では、ファイルシステムのディレクトリ構造が変更されている。これは、ネットワークファイルシステムによって、ディレクトリが共有されることを意識したものになっている。4.3BSD-Reno のファイル構造を図 2.1 に簡単に示す。

```
/
    bin/                シングルユーザコマンド
    sbin/               管理用コマンド
    dev/                デバイスファイル
    etc/                管理用テキストファイル
    root/               root のホームディレクトリ
    tmp/                テンポラリ
    usr/bin/            コマンド
        sbin/           管理用コマンド
        lib/            ライブラリ
        libdata/        ライブラリデータ
        libexec/        ユーザが直接利用しないコマンド
        share/          アーキテクチャに依存しない共有データ
    var/                計算機に固有のデータ
```

図 2.1: 4.3BSD-Reno のディレクトリ構造

また、ソースファイルも異なるアーキテクチャの計算機間で共有が可能である。make コマンドが拡張され、オブジェクトファイルを別の計算機に固有なディレクトリに作成するようになって、Makefile の中で実行時に処理を変更するような仕掛けが入っている。

ルートファイルシステムは可能な限り小さくなるように構成され、アーキテクチャの同じ計算機間で共有可能なものになる。また、起動時には読み込みのみでマウントされ、マルチユーザに移行する時に書き込み可能にされる。

特定の計算機に固有のデータは /var の下にまとめられる。

2.2.2　ISO/OSI ネットワーク

4.3BSD-Reno のもっとも大きな特徴の一つが ISO/OSI ネットワークのサポートである。アプリケーション層は ISO の開発環境である ISODE が取り入れられている。トランスポートおよびネットワーク層はウィスコンシン大学の実装が採用され、この中にはトランスポートクラス 4 (TP-4)、コネクションレスネットワークプロトコル (CLNP)、End System-Intermediate System プロトコル (ES-IS) などが含まれる。

OSI プロトコルを実現するために、4.3BSD では固定長 (14bytes) であった sockaddr 構造体中のアドレスフィールドは不定長になった。それにともなって経路情報のテーブルのアドレスも不定長になり、またラウンドトリップタイム (RTT)、ホップ数、パケット長を経路情報として持つ。

ネットワークデータの受渡しに使われる mbuf の構造もかなり変更された。従来 mbuf 専用のメモリプールから確保されていたのが汎用のメモリ管理インタフェースによって確保されるようになった。チェーンの先頭と、それに続く mbuf ではメモリの利用の方法が異なり、効率的に管理されるようになった。構造体のアドレスからのオフセットによっ

て指示されていたデータの格納領域は、実際のアドレスを持つように変更された。これらの変更によって、ネットワークハードウェアとのインタフェースがかなり影響を受けている。

2.2.3 TCP/IP ネットワーク

Tahoe のリリース移行に行なわれたいくつかの TCP/IP ネットワークに関する改良も含まれている。

ヘッダ予測 (Header Prediction) TCP プロトコルで次に返ってくるパケットのヘッダをあらかじめ予測することによってパケットの受信処理を高速に行なう機構が導入された。

CSLIP ヘッダを圧縮することによって実際にネットワークにながれるデータの量を減らすことによって高速化を行なう header compression が採用され、シリアルラインによるネットワークが高速化された。

この他、経路制御に関する情報が増え階層的な制御が可能である等の変更が含まれる。

2.2.4 NFS

Sun Microsystems 社によって開発された NFS (Network File System) の機能が追加された。そのために複数のファイルシステムを切替えるための機構である VFS (Virtual File System) インタフェースが導入された。

NFS プロトコルは RPC/XDR の上に実装されるが、VFS と RPC/XDR に関するコードは Sun Microsystems 社から提供されたものを利用している。実際の NFS のサーバ/クライアントの実装は Sun のものとは無関係であり、Sun からのライセンスフリーなコードを利用している。

NFS プロトコルはバージョン 2.0 を使用し、これは現在普及している NFS の実装と互いに通信することが可能である。また、従来と同じ UDP 上に実装された NFS プロトコルに加えて TCP 上でも NFS が利用できる。これによってシリアルラインなどの比較的低速なネットワークを介しても NFS が利用できる。

NFS はサーバマシンのディレクトリをマウントすることによってリモートのファイルの操作が可能になるが、このマウント時のオプションに soft, hard に加えて spongy という選択が可能である。これは soft と hard の中間に当たるもので、stat() や open() に関しては soft マウント (タイムアウトする) であり、read(), write() に関しては hard マウント (タイムアウトはしない) であるように振舞う。

BSD 上の NFS の実装は、単に標準的な NFS が利用できるというだけではなく、これからのファイルシステムインタフェースの統一に向けての研究の第一歩である [112]。

2.2.5　汎用 malloc

従来データタイプ毎に用意されていた動的なメモリ管理システムを統一し、汎用的なメモリアロケータが用意されている。これにより効率的なメモリの利用が可能である。また、この機能は十分高速に働くよう実装されているので、従来に比べて速度的に不利な点はない [113]。

2.2.6　QUOTA

QUOTA システムは、ユーザとグループによる管理が可能なように変更され、従来ログイン時に行なっていたチェックを時間で管理できるようになった。

2.2.7　MFS

MFS は VFS の下に実現されたメモリファイルシステムである。多くのオペレーティングシステムでメモリ上にファイルシステムの実装が行なわれているが、これらはファイルシステムのアクセスは高速に行なわれる代わりに、ファイルシステムの全体的な大きさが制限されるという欠点があった。MFS ではこれを解決するために、仮想記憶上にファイルシステムを実現するという方法をとっている。これによって、大きさに制限されない高速なファイルシステムが利用可能である。MFS は主に /tmp に利用される [114]。

2.2.8　POSIX

4.4BSD では IEEE POSIX P1003 シリーズで定義される標準化の参照モデルを提供することを目標の一つとしており、その一部が 4.3BSD-Reno で実装されている。主に関係する部分を以下に示す。

1003.1 1003.1 はオペレーティングシステム及びライブラリのインタフェースである。4.4BSD では、この完全な実装を目標としており、現在は 90% 程度が実現されている。

1003.2 シェルとユーティリティプログラムに関するインタフェースである。

1003.12 ネットワークインタフェースである。まだ数多くの問題が残されている。

POSIX インタフェースの中でもっとも大きなものは端末インタフェースの変更である。基本的なインタフェースは System V と互換性を持つように作られ、それに 4.3BSD の端末ドライバとの上位互換性を保つための機能が追加されている。

また、POSIX に準拠するジョブコントロールの機能が取り入れられた。これは 4.3BSD に似た仕様であるが、安全性の高いプロセス管理を行なえるような仕組みになっている。

POSIX で定義されるシグナルは基本的に 4.2/4.3BSD から取り入れられたものであるが、若干の修正が入っている。

ライブラリ関数は約 80% が ANSI/C 準拠となっている。4.4BSD では 100% の対応が行なわれる予定である。

2.2.9 Kerberos 認証システム

MIT で開発された Kerberos 認証システムのバージョン 4 が組み込まれている。これによって、信頼性の高いユーザの認証機構が実現されるが、アメリカ以外の国に対する配布の中には暗号化のプログラムの一部が含まれているので、配布ソフトウェアだけで Kerberos システムを利用することはできない。また、login などの Kerberos を利用するユーティリティプログラムは、Kerberos を使わずに作成されるように変更されている。

4.3BSD-Reno で Kerberos 対応になっているコマンドは以下の通り。

- bin/rcp
- libexec/rlogind
- libexec/rshd
- libexec/kpasswdd
- usr.bin/rsh
- usr.bin/login
- usr.bin/rlogin
- usr.bin/telnet
- usr.bin/passwd
- usr.bin/su

また、パスワードファイルには shadow password という機構が採用されている。これは /etc/passwd ファイルに直接暗号化したパスワードが書かれていずに、アクセスする権限を持ったプログラムだけが暗号化された文字列を参照できる仕掛けである。これによってパスワードを破るようなプログラムを作成することは困難になる。また、パスワードの有効期限を設定することができる。

2.2.10 カーネルトレース

次のような項目に関してカーネルの動きをトレースする仕組みが組み込まれている。

- システムコール
- パス名の解釈

- シグナル処理
- 入出力

また、これらのトレースを行なう対象を指定することができる。

- 任意の実行中のプロセス
- コマンド
- プロセスグループ
- 子どものプロセス全部

2.2.11　その他

カーネルのシステムコールの入口が、通常の関数呼びだしの形に統一された。例えば、4.3BSD では read() システムコールの定義は、

```
read()
{
        register struct a {
                int     fdes;
                char    *cbuf;
                unsigned count;
        } *uap = (struct a *)u.u_ap;
...
}
```

のような形で引数を受けとっていたが、4.3BSD-Reno では次のように通常の引数として受けとることができる。

```
read(p, uap, retval)
        struct proc *p;
        register struct args {
                int     fdes;
                char    *cbuf;
                unsigned count;
        } *uap;
        int *retval;
{
...
}
```

また、user 構造体の中のデータとして返していたエラー番号は、関数の戻り値として返されるようになった。それにともない、setjmp, logjmp の使用もなくなり、カーネルプログラムの可読性が向上している。

2.3 4.4BSD

4.3BSD-Reno には 4.4BSD でサポートされる予定の機能のいくつかが含まれていない。

仮想記憶システム 4.4BSD では、CMU で開発された MACH の仮想記憶を採用する予定である [115]。

ISO 関連 ISO のコネクションオリエンテッドネットワークサービス、X.25、TP-0 のサービスがサポートされる予定である。

ストリーム 4.4BSD では Bstreams と呼ばれる、ストリームとソケットを融合するための新しいネットワーク機能が取り入れられる予定である。

これらのいくつかの大きな機能がサポートされていないため、4.3BSD-Reno は 4.4BSD-alpha リリースという位置付けになっていない。

また、WIDE プロジェクト以外でも次のようないくつかアーキテクチャに対するの移植作業が進んでいる。

- MIPS アーキテクチャ (R2000, R3000, R6000)
- Sun (MC680x0, SPARC)
- Intel 80386

これらの一部あるいは全部が 4.4BSD ではサポートされる。

第 3 章

新しいオペレーティングシステム技術

3.1 Mach オペレーティングシステム

今日の計算機環境はネットワークの発展にともない、多くの計算機が接続された広域分散環境となり、資源の共有や遠隔地との情報交換が行なわれている。資源の利用や情報の伝搬は、ネットワークに接続された各計算機に用意された様々なアプリケーションプログラムを利用することにより行なわれている。これらのプログラムがネットワークを介して別の計算機と通信を行ない、ユーザに様々なサービスを提供している。これらのプログラムは広域分散環境には欠かせないものであり、今後も新たなネットワークサービスのために開発されると推察できる。これら広域分散環境で利用されるアプリケーションプログラムの数が増加することを考えると、ネットワークをより効率的に使用するためには、これらのプログラムのより効率的な実装が重要であると考えられる。

また最近、従来までの UNIX の機能を継承しながら、ネットワークやマルチプロセッサなどの分散環境を強く意識して、改良や拡張を加え構築された新しいオペレーティングシステムが現れている。UNIX との互換性を保っているので、従来の UNIX と同様の使い方をすることも可能である。しかし従来通りの方法では、改良されたオペレーティングシステムの機能を有効に使っているとは言えない。これらの機能を有効に使えれば、従来と比較して大きな効果を得ることが可能となる。この様な新しい機能は近年様々な形で普及してきており、多くの計算機上で利用できるようになっているが、その有用性の確認にはまだまだ多くの議論が必要である。

本論文では、一つの使用方法として、ネットワークアプリケーションの効率的な実装を行ない、新しい機能の利用を試みる。

3.2 目的

現在のネットワーク環境で使われるアプリケーションプログラムでは、プロセス間通信を利用した、クライアント・サーバモデルと呼ばれる方法がよく利用される。現在の UNIX システムで動作しているサーバプログラムは、クライアントからの要求に対し、サーバプロセスの複製を作成し処理を行なっている。

また最近、UNIX をベースに開発された Mach オペレーティングシステムにはプロセス

に代わる概念としてタスクとスレッドがある。サーバのようなプロセスの操作を行なうプログラムを考えると、このタスクとスレッドが利用できればより効果的な実装が可能になると考えられる。本研究では Mach オペレーティングシステムのスレッドを利用して、サーバプログラムの効率的な実装を試み、その性能や問題点について検討する。またサーバプログラムに限らず、この様な手法によって得られる効果を研究し、応用的な利用方法とその問題点について考察する。

以下では、まず第 4章でクライアント・サーバモデルと Mach について述べ、スレッドを利用したサーバプログラムについて考える。そして第 5章で実際にサーバプログラムを作成し、評価、考察を行なう。

第 4 章

背景

4.1 分散環境

複数の計算機が接続された分散環境では、ネットワーク内の資源を利用するために様々なアプリケーションが用意されている。これらの分散環境型アプリケーションの多くは、その機能を実現するために複数のプログラムを使い、それらがプロセス間通信を行って、役割を分担しながら実際の処理を行なっている。

4.1.1 クライアント・サーバモデル

プロセス間通信を利用したプログラムを作成する場合の一般的なモデルが、クライアント・サーバモデルである。(図 4.1)

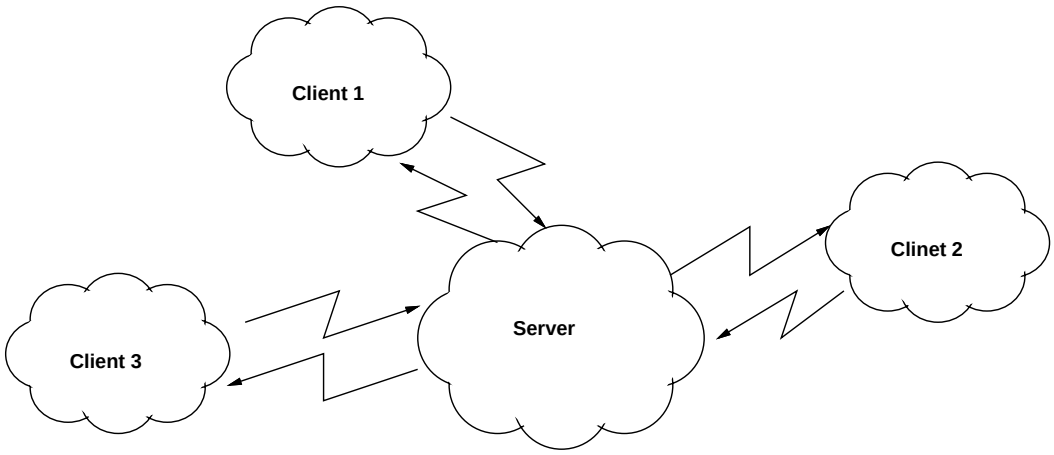

図 4.1: クライアント・サーバモデル

クライアントはユーザがサービスを利用するために使用するプログラムで、そのサービスに対応するサーバへの要求を行ない、その結果をユーザに返す。

サーバは実際のサービスを提供するプログラムで、計算機の稼働している間は常時システム内に存在する。サーバは起動されると自分の提供するサービスに必要な初期化処理の後、クライアントからの要求待ちの状態になる。要求を受けとると、それに応じた

処理を行ない、その結果をクライアントに送り返す。またサーバがそのサービスに関する処理をすべて引き受け、資源の利用と管理を行ない、競合や矛盾が生じないための処理も行なう。

例えば **lpd** というプリンタサーバは、ユーザからの出力要求を受けた順番に従ってプリントアウトを行なう。プリンタへのアクセスが無秩序に行なわれれば、出力される順番は保証されず、正しい結果を得ることは出来ない。

通常サーバは複数のクライアントからの要求を同時に処理できるように作成される。現在の UNIX システム上でのサーバプログラムでは、複数の要求を処理する方法に次の 2 つがある。(図 4.2)

1. 1 つのサーバプロセスですべてのクライアントの処理を行なう。
2. 各クライアントごとに 1 つのサーバプロセスを対応させて処理を行なう。

実際のサーバプログラムがどの様な方法で実装されているかは、サービスの種類によって異なる。

2の方法を使用するのは以下のような理由がある。

- クライアントの要求を満たすのに時間がかかり、他のクライアントからの要求に答えられなくなってしまう場合。処理中に別のクライアントからの要求を検査するのは困難である。
- サーバ側がサービスの最中に様々な状態を遷移するようなサービス。複数のクライアントに対する状態を 1 つのプロセスで管理するよりは、個々に対応をした方がプログラムが簡単になる。

逆に 1の方法は、上で述べた条件があてはまらず、多くの場合は処理が簡単なサービスを提供するサーバである。[1]

サーバの起動

サーバを起動する方法には大きく分けて 2 つの方法がある。

1. システムの起動時に、提供するサービスに対応したサーバをすべて起動し、要求待ちの状態とする。
2. すべてのサービスに対する要求を受け取り、要求を受けてからそれに応じたサーバプロセスを起動する。

[1] 本研究では主に 2の場合について考えることにするが、1 プロセスでサービスを行なう場合にも応用は可能であると考えられる。

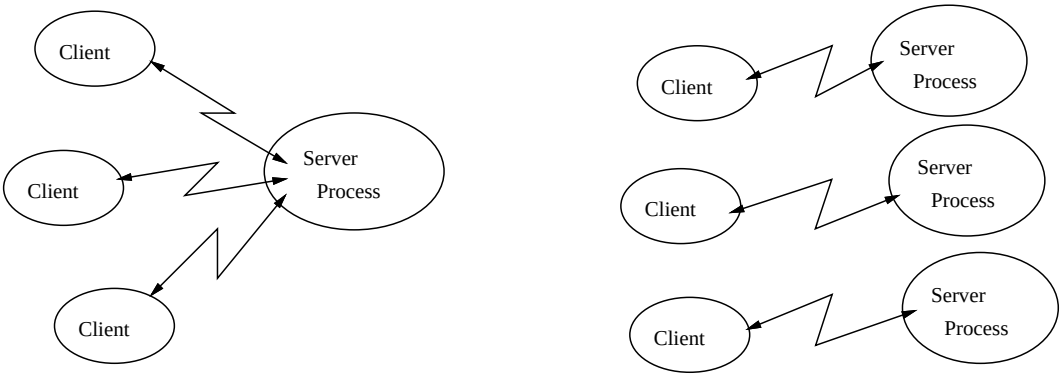

図 4.2: クライアントとサーバの対応

1の方法では、サービスの数と同数のプロセスを起動する必要があり、サービスが多い場合に計算機に負荷をかけることになる。

そこで 4.3BSD では、**inetd** と呼ばれるプログラムを使い 2の方法を行なう。この方法では要求待ちを行なうプロセスの数が減り負荷は軽減されるが、要求を受けてからプロセスを起動するため、サービスの開始時にオーバヘッドが生じる。

4.1.2　遠隔手続き呼び出し

クライアント・サーバモデルによるプログラムでは、クライアント・サーバ間の通信路やプロトコルの設定が必要である。しかしこれらの作業を統一的にし、利用できるようにしたのが遠隔手続き呼び出し（Remote Procedure Call）である。

遠隔手続き呼び出しでは、ネットワーク機能を提供する関数が用意されており、プログラムから普通の関数と同様に呼び出すことが出来る。クライアントが行なった遠隔手続き呼び出しは、サーバへのサービス要求として送られる。サーバは要求に応じた処理を行ない結果を返す。

遠隔手続き呼び出しではプロセス間通信を意識せずに、一般の関数呼び出しと同様に扱えるため、プログラムの構築が容易となる。しかし、そのためにはネットワークサービスを行なう関数を用意する必要がある。この様なネットワークライブラリの拡充が課題である。

4.2 Mach

Mach Operating System は米国カーネギーメロン大学で開発された UNIX 互換のオペレーティングシステムである。カルフォルニア大学バークレー校で開発された UNIX 4.3BSD をベースに新しい技術を取り入れている。

Mach の特徴を以下に挙げる。

- カーネルは小さく、ユーザから見える機能はあまり多くない。OS としてのいくつかの機能はユーザプログラムとして用意され、カーネルの外で動作する。
- カーネル機能の実装をカーネル外のプログラムにしたことにより、ユーザプログラムで様々な機能の実装を可能とし拡張性を高めた。
- マルチプロセッサシステム、大容量メモリや高速ネットワークに対応している。

また Mach カーネルの提供する機能は、表 4.1に示したもので極めて基本的なものである。Mach ではこれらの機能を利用するためのカーネルインターフェイスを用意している。

オペレーティングシステムとして UNIX と同等の機能を実現するのは、カーネルプリミティブを使って UNIX のエミュレーションを行なうプログラムの機能である。通常のユーザアプリケーションプログラムと UNIX エミュレーション部とのインタフェースは、従来の UNIX と同様にして互換性を保っている。

その一方で、ユーザアプリケーションが直接 Mach カーネルインターフェイスを用いることが可能であり、Mach の機能を有効に使ったプログラムを作成することも可能である。（図 4.3）

計算とメモリの管理	
タスク	実行環境であり、資源の割り当てを管理する単位でシステム資源（仮想アドレスなど）の保護を行なう。
スレッド	実行の基本単位であり、実行に必要なプロセッサ状態（ハードウエアレジスタ）など保持する。
メモリオブジェクト	メモリ管理の単位で、領域の割り当て、解除、保護を管理する。
通信	
ポート	送信または受信の単方向の通信チャンネルで、メッセージキューとして実装される。
メッセージ	型別のデータオブジェクトの集合で、スレッド間の通信に使われる。

表 4.1: Mach カーネルの概要

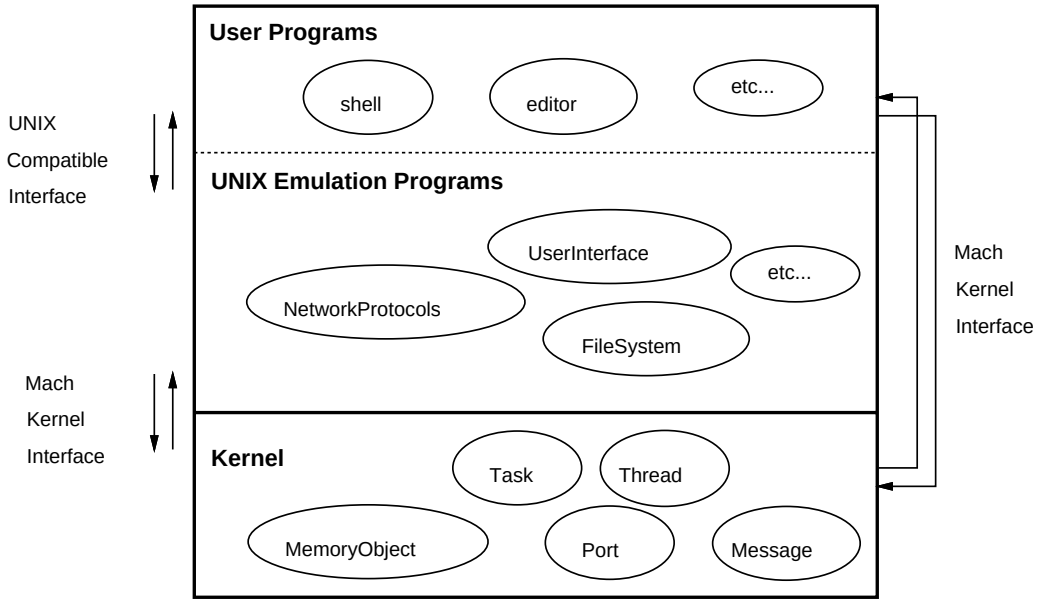

図 4.3: Mach の構成

4.2.1　タスクとスレッド

Mach では従来のプロセスの概念をタスクとスレッドの 2 つに分けて考える。

タスク アドレス空間などの資源を管理し、実行に必要な環境を保持する。複数個のスレッドを持つことができる。

スレッド プログラムカウンタやレジスタなどの実行に関連する情報を持ち、1 つのタスク内に存在する。

タスク内のすべてのスレッドはタスクの持つ資源にアクセスすることができる。つまりタスク内の資源はすべてのスレッドで共有していることになる。スレッドはスケジューリングの単位であり、スレッド単位で CPU 時間が割り当てられる。1 つのタスクは複数のスレッドを持つことができ、同一タスク内の複数のスレッドは同時に実行することができる。1 つのタスク内に 1 つのスレッドが存在する場合を考えれば、従来の単純なプロセスと同等になる。（図 4.4）

1 つのタスク内で複数のスレッドを動かす際に注意しなければならないことは、共有している資源（メモリなど）に対する非同期のアクセスである。タスク内の各スレッドが並列に動作するため、これらの資源についてユーザが排他的制御を行なう必要が生じる。

C-Thread

カーネルインターフェイスを直接的に使用してアプリケーションプログラムを作成するのは容易ではない。そこで C 言語 [116] から利用できるライブラリ（C-Thread Library）が提供されている。このライブラリは多重スレッドの操作、相互排除機構によるクリティ

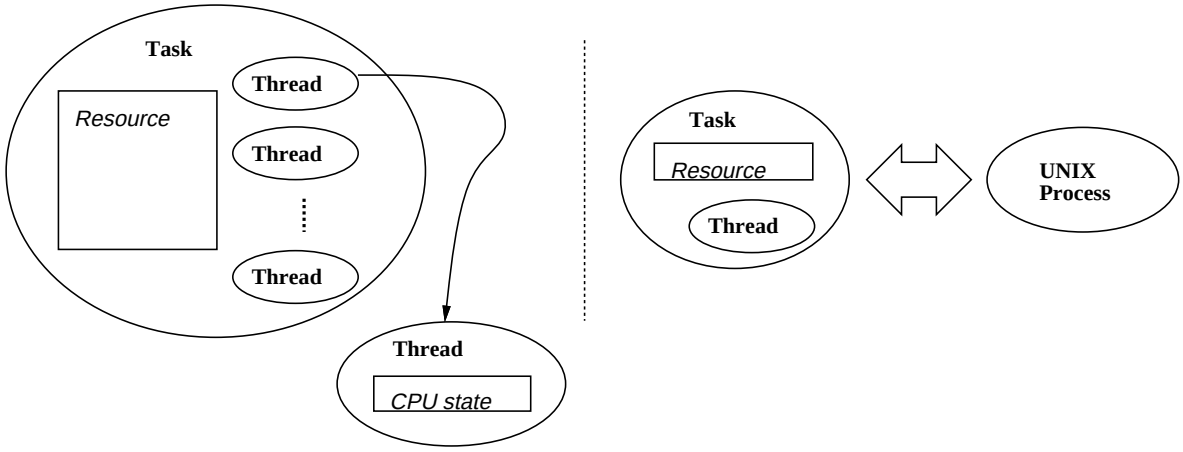

図 4.4: タスクとスレッド

カルセクションの保護、条件変数を使った同期化などの機能を提供している。

C プログラムが開始される時点ではスレッドは 1 つしかなく、関数 main から実行が始まる。新しく作成されるスレッドは指定した関数から実行を始める。各スレッドは実行を開始した再上位の関数から戻る時に終了するが、明示的に終了することも出来る。

共有データに対する保護は、相互排除プリミティブを使用する。共有データにアクセスするスレッドは、まずプリミティブをロックしてからアクセスを行ない、アクセス後にロック解除を行なう。ロックを試みる時に、すでに他のスレッドによりロックされていた場合はロックが解除されるまで、そのスレッドはブロックする。複数のスレッドが同時にロックを試みた場合には、一つのスレッドがロックに成功し、他のスレッドがブロックすることを保証している。

条件変数はスレッドが他のスレッドの処理を待つ場合などに使われ、相互排除プリミティブによって保護されている。一般的な使用方法は、以下の様になる。

1. スレッドは相互排除プリミティブのロックを行ない、共有データの検査を行なう。

2. 検査の結果、条件が成立してないため他のスレッドの処理を行ないたい場合がある。この時、一時的にロックを解除し、他のスレッドが共有データにアクセスできるようにし、条件変数を指定して待機状態になる。

3. 他のスレッドは共有データを変更し、条件変数とともに通知する。その結果、2で待機していたスレッドがロックを回復し処理を再開する。

4. 再開したスレッドは再度条件の検査を行い、満たされるまで 2を繰り返す。

4.2.2　仮想アドレス空間

仮想アドレス空間は、固定サイズの仮想ページの集合である。ページのサイズは計算機に依存している。仮想アドレス空間はタスクに結びつけられて、タスクによって操作される。

タスクは仮想アドレス空間への仮想ページの割り当て、既に割り当てられた空間内のページの解放、などの操作を行なう。仮想ページを仮想アドレス空間に割り当てる時に、物理的な資源が確保されるのではない。物理的な資源の割り当てが実行されるのは、仮想ページの割り当てられたアドレス空間を初めてアクセスした時点である（map on reference）。（図 4.5）

タスクが新たなタスクを生成する際に、親タスクの持つ仮想アドレス空間を子タスクに継承することができる。継承は仮想ページ単位で行なわれ、各ページ毎に以下のような指定が可能である。

なし 子タスクに継承しない。子タスクの該当するアドレス空間は割り当てられない。

複写 仮想ページは複写され、各タスクが行なった変更は独自のものとなる。

共有 親子のタスクで共有する。片方のタスクが行なった変更を、他方のタスクで検出できる。

複写の場合、実際に仮想ページの複写を行なうのは、そのページに変更が加えられる時であり、それまではページを共有している（copy on write）。（図 4.6）

以上の様に必要になるまでは、実際の資源を割り当てずにおくことで、システムの資源を有効に利用している。

4.2.3　ポートとメッセージ

ポートは保護されたカーネルオブジェクトであり、メッセージは固定長のヘッダと可変長のメッセージ本体からなるデータ集合である。

ポートへ送信されたメッセージは、キューに保存される。メッセージはポートの受信権（後述）を持つタスク内のスレッドによって受信される。受信されたメッセージはキューから取り除かれる。（図 4.7）

ポートへのアクセス権は、送信権、受信権、所有権からなる。タスク内に新しくポートを作成すると、そのタスクが 3 つの権限を獲得する。

送信権 そのポートへメッセージを送信できることを示す。複数のタスクが 1 つのポートへの送信権は持つことが出来る。

受信権 そのポートへのメッセージを受信できることを示す。1 つのタスクのみがポートへの受信権を持つことが出来る。受信権には必然的に送信権を伴う。

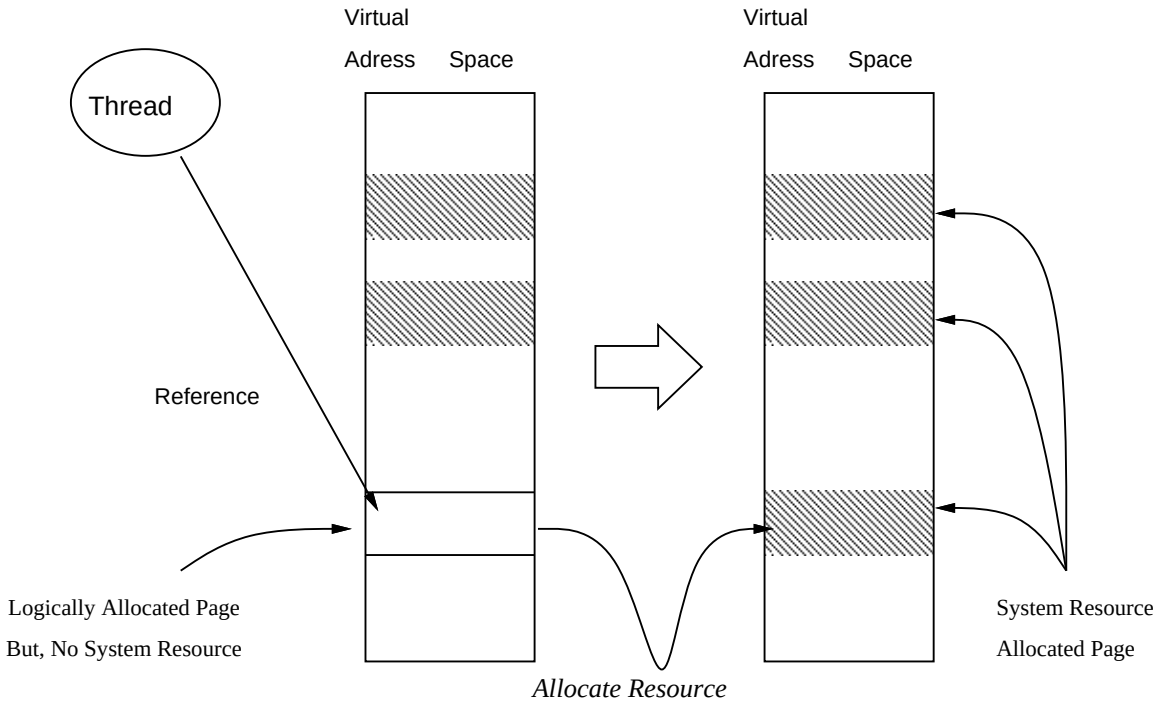

図 4.5: 参照時割り当て

所有権 そのポートの受信権を持つタスクが、受信権を破棄した場合に、その受信権を受け取ることが出来る。逆に、所有権を持つタスクが所有権を破棄した場合には、受信権を持つタスクが、その所有権を受け取る。1 つのタスクのみがポートの所有権を持つ。所有権には必然的に送信権を伴う。

ポートの権限は他のタスクへ受け渡すこと可能である。受信権と所有権の両者を持つタスクが終了すると、ポートは破棄される。

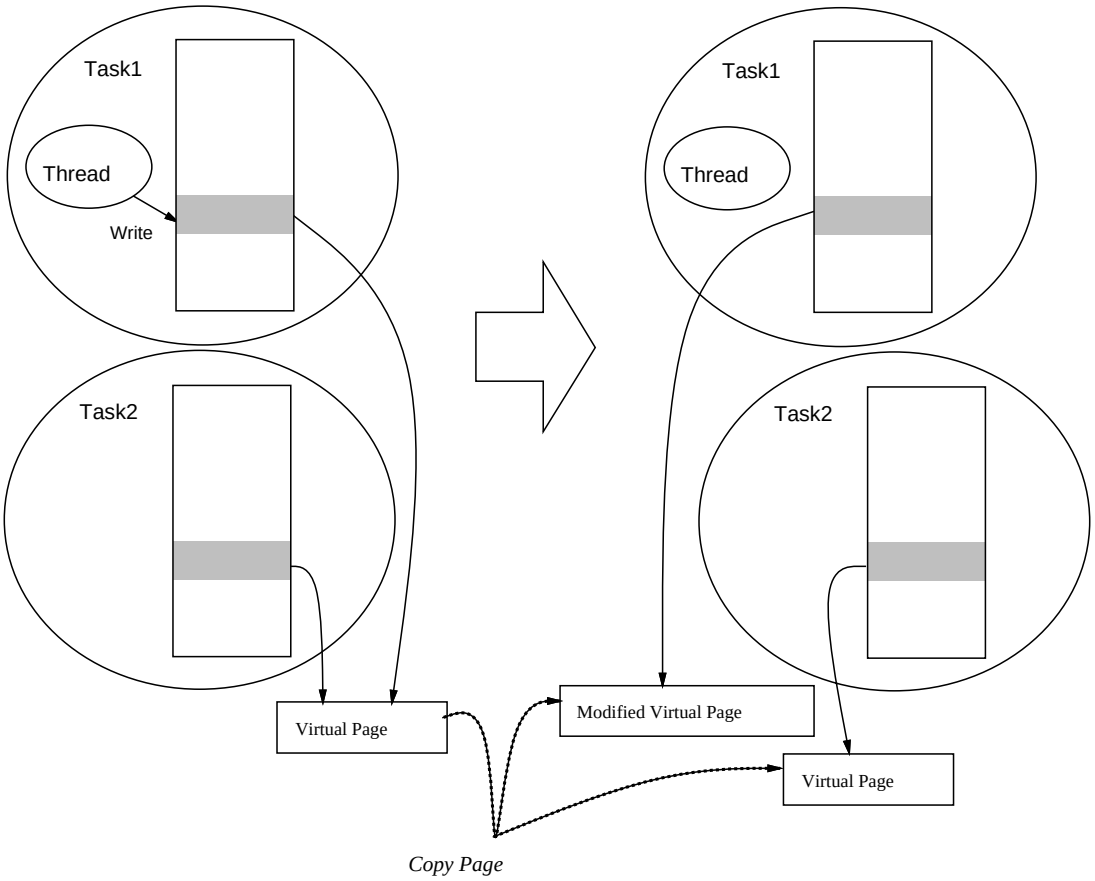

図 4.6: 書き込時複写

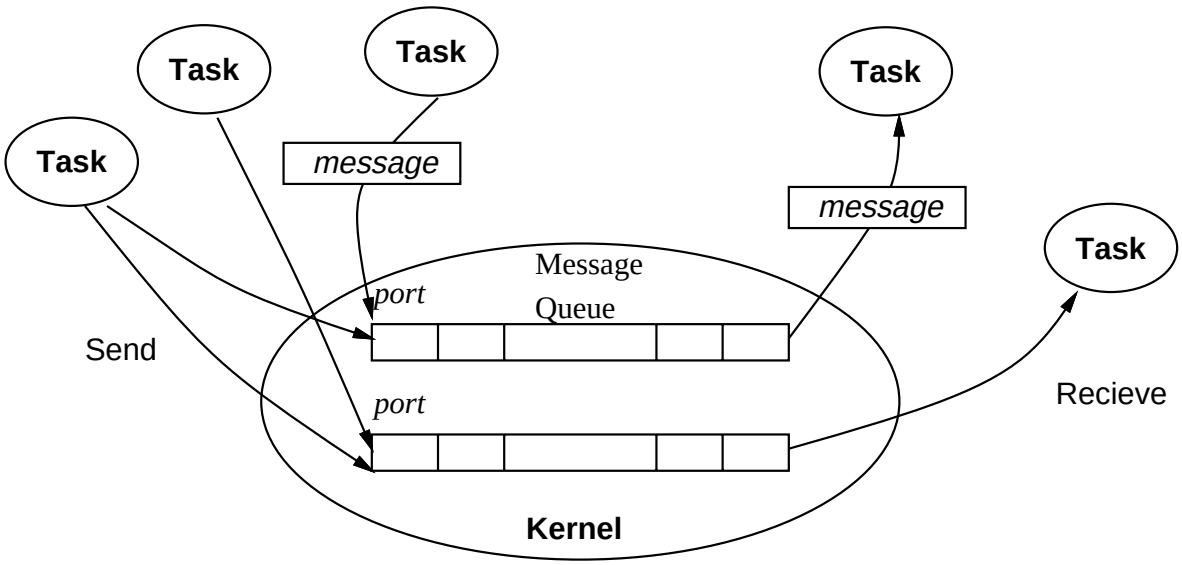

図 4.7: ポートとメッセージ

4.3 UNIX におけるクライアント・サーバモデルの問題点

クライアント・サーバモデルの一般的な実装は以下のようになる（図 4.8）。

- サーバプロセスは初期化処理を行ない要求待ちになる。[2]
- クライアントからの要求を受け取ると、自分自身の複製となるプロセスを作成する。
- そのプロセスがクライアントとの通信を行ないサービスを提供する。
- 元のサーバプロセスは他のクライアントからの要求を待つ状態に戻り、要求が来ると再度複製を作成しサービスを行なう。
- 複製されたプロセスはサービスが完了した時点で終了する

これらの方法を考えた場合、以下に述べる点はスレッドを利用することでより効果的に処理できると考えられる。

サーバ内の情報

サーバ同士が内部的に共有する情報の管理が難しい。実際の処理を行なうプロセスは複数存在できるので、それらのプロセス同士が競合や矛盾を起こさないようにしなければならない。従来はサーバ同士がプロセス間通信を行なう、ファイルシステムを利用する、共有メモリを使用する等の方法が考えられた。

スレッドを使用する場合、サーバタスク内の情報はすべてのスレッドから参照できることを利用し、タスク内で共通な情報の管理を行ない、各スレッドはそれらの情報を利用して処理を行なえばよいことになる。

仮想メモリ

プロセスの生成により、新たな記憶領域が必要となる。テキスト領域は各プロセス共通なので共有することができるが、データ及びスタック領域は各プロセスごとに用意しなければならない。

これに対しスレッドの場合では、テキストとグローバルデータは共有されており、ローカルデータのためにスタック領域を必要とするだけである。

プロセスの実行

システム内では複数のプロセスが実行されている。実行プロセスを切替える（context switching）という操作はシステムにとって主たる負荷のひとつである。プロセス数の増加はシステムの負荷を上昇させる要因となる。

プロセスの切替えには

[2] 4.3BSD では **inetd** が要求を受けとり、処理に応じたサーバを作成する。

1. 今まで実行していたプロセスの状態（プログラムカウンタ、レジスタなど）を保存する。
2. 次にスケジューリングされたプロセスのために仮想アドレス空間の切替を行う。
3. 保存しておいたプロセスの状態を取り出し、実行可能とする。

などの手順が必要であるが、もっとも負荷がかかるのは仮想アドレス空間に対する処理である。

タスク内のスレッドは同じ仮想アドレス空間に属するため、同一タスク内のスレッドでコンテキストスイッチが起きた場合などに切替を必要とせず、負荷の軽減が期待できる。

サーバプログラムを複数のプロセスで行なう場合（図 4.9）と、複数のスレッドで行なう場合（図 4.10）は図の様になる。

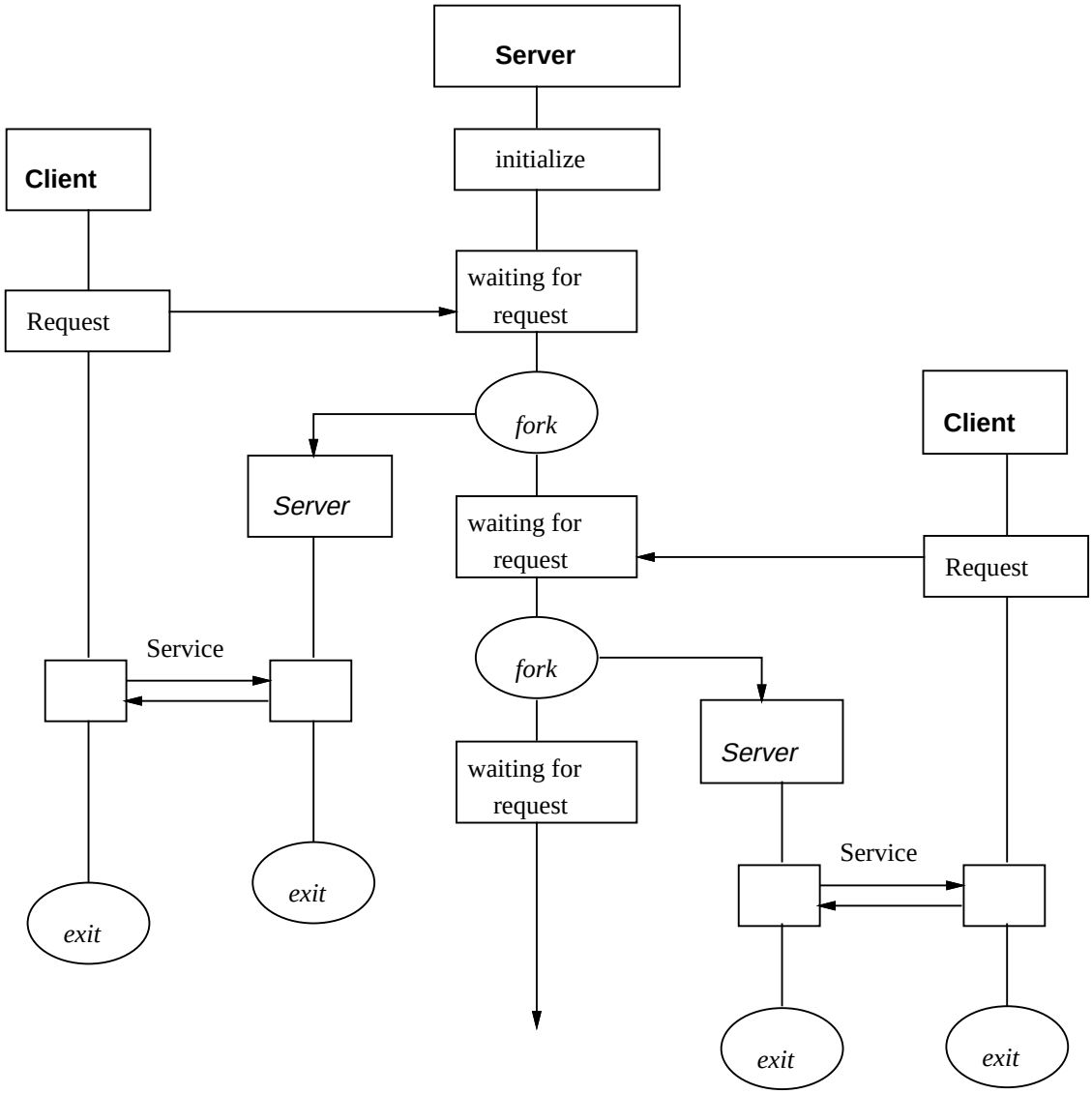

図 4.8: クライアント・サーバモデルの構造

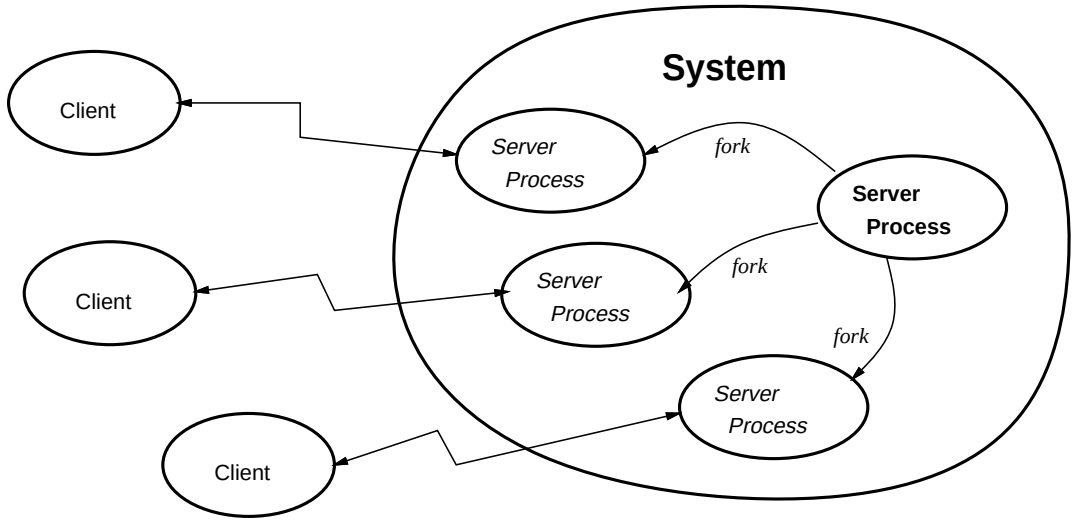

図 4.9: マルチプロセスによるサーバ

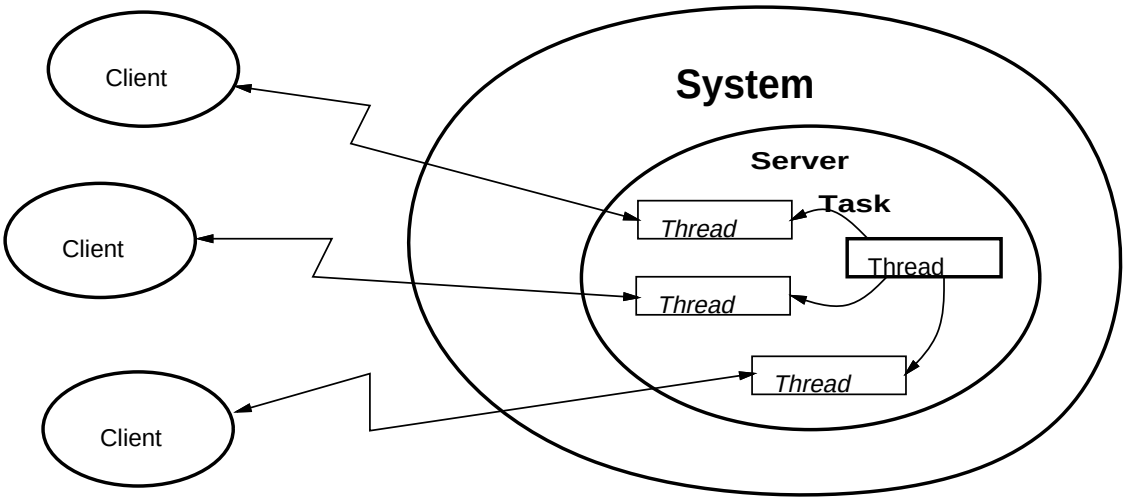

図 4.10: マルチスレッドによるサーバ

第 5 章

Multi Threaded Server の作成

これまでの議論からスレッドを使用したサーバプログラムの作成を行ない、スレッドを利用することの利点と欠点との考察を行なう。どの様に使用すれば、より効果的に利用できるかを考える。

5.1 設計

今回作成するサーバプログラムは、タスクとスレッドの機能を利用することが目的である。そこで以下の点を目的に作成する。

- 現在のサーバプログラムではクライアントの要求に対し、サーバプロセスの複製をつくりサービスを行なっているが、複数のプロセスに代わり、複数のスレッドで処理を行なう。
- サーバプロセス同士で共有すべき情報の管理をタスク内で行ない、各スレッドから参照を行なう。

以上の目的から、スレッドを利用したサーバプログラムを作成するが、本研究では簡単なデータベースを扱うサービスを考えた。
ここで考えるサービスは以下に述べる様なものである。

- サーバ側の計算機でデータベースファイルの管理を行う。クライアントからの要求に応じてデータベースへの登録削除および参照を行ない、その結果をクライアントに返す。
- サーバは要求を受け取ると新しいスレッドを作成し、以降のサービスはそのスレッドが行なうこととする。
- タスク内の各スレッドはデータベースファイルへのアクセスを行なうが、データベースファイルの情報を共有している。アクセスを行なう時はスレッド間で相互排除を行い、同時にアクセスできるのは 1 つのスレッドとなることを保証する。

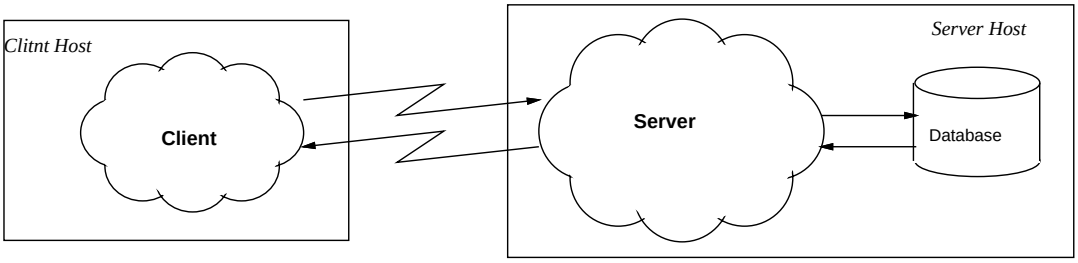

図 5.1: データベースサーバ

5.2　スレッドの限界

現在クライアント・サーバモデルで実現されているサービスは様々なものがあり、またサービスの性質も様々である。サービスの種類によってはスレッドを利用することができないものがあり、また利点がないものがあると考えられる。

別のプロセスをサーバ側のホストに作りだし、実行することをサービスとするようなアプリケーション、例えば rsh や rlogin などの場合は、サーバプロセスの複製が exec システムコールで目的とするプロセスを実行することで実現されている。サービスの最終的な目的がプロセスの生成、つまりタスクの生成であるので、これらのアプリケーションでスレッドを使用する必要はなく従来通りの方法を使うことになる。

また実行時権限を考慮すべきアプリケーションでは、スレッドで処理を行なうのに向いているとは考えられない。何故なら、サーバプロセスは新たな複製プロセスを生成する時に、そのプロセスの権限を変更することが出来る。この様に複製されたサーバプロセスは変更後の権限で実行されるので、権限に応じた保護の下でサービスを提供することになる。スレッドには権限はなくタスクが権限を持っているため、同一タスク内のスレッドが提供できるサービスは同一の権限内のものに制限される。サーバプログラム内で各スレッドに権限に相当するデータを持たせ、プログラム側でその権限を確認することも可能ではある。しかし、プログラムが複雑になり様々な危険が生じる可能性があるため、利点を得ることはできないと考えられる。

5.3　実装

実装は OMRON LUNA-88K の Uni-OS Mach 2.5 上で、C-Thread パッケージを使用して行なった。サーバ・クライアント間の接続はソケット（**socket**）と呼ばれるプロセス間通信機能を使う。これは、TCP/IP[5] で通信を行い信頼性がある。

またスレッドを利用した場合との比較のため、従来のクライアント・サーバ方式のプログラムも作成する。

サーバタスクの処理の概略を示す。

- サーバが起動されると初期化処理の後、クライアントからの接続要求を待つ。
- クライアントから要求を受け通信路が確立したら、新しくスレッドを作成し、以降のサービスは新しく作成されたスレッドが行なう。元のスレッドは要求待ちに戻る。
- 各スレッドはサービスが完了したら終了する。

各スレッドがデータベースへのアクセスを行なう時には、C-Thread の同期機構を用いて排他制御を行ない、同時に複数のスレッドがアクセスするのを防ぐ。

5.4 評価

5.4.1 プログラム作成

スレッドを使う場合は、複数のスレッドが同時に処理を行なう可能性を常に注意する必要がある。

従来のプログラムでは、fork システムコールを用いて複製プロセスを作成する。しかしすでに存在するサーバプログラムを変更して、スレッドの使用を考えた場合、プロセスの作成をスレッドの作成に変更するだけでは十分ではない。

プログラム内の広域変数や関数内で保持される静的変数に対する参照、変更を行なう場合は、相互排除を行なうようにしデータの破壊が起きないように注意する。

これらの点を考慮し作成することは、従来のサーバプログラムを作成する場合に比べ、それほど大きな負担とはならない。逆に、タスク内での情報の共有をうまく利用すれば、従来のプログラム作成に比べて容易に作成が可能である。

プログラム作成時の一番大きな問題点は、プログラムのエラー検出である。多重スレッドの場合、タスク内のデータが非同期にアクセスされるため、実際にエラーを起こしたスレッドと、エラーの原因となる動作を行なったスレッドが必ずしも同じではない。

排他制御に誤りがあると、この様な事が頻繁に起きる可能性が高く、プログラムの動作に異常があった場合に原因を特定することが困難となる。通常、エラー検出にはデバッガを使用するが、現在使用されているデバッガでは多重スレッドに対応していないため、効果的な利用が出来ない。

5.4.2 情報の共有

タスク内で情報を共有することで、すべての情報は通常のデータと何ら変わることなくアクセスできるため、各スレッド間の情報の伝達は非常に簡単になった。

しかし共有する情報が増えると、競合を避けるための排他制御が複雑になる。プログラム作成者に若干の負担がかかるため、状況に応じて選択することになる。

5.4.3　メモリ資源の消費量

仮想メモリの使用量を考える。1 つのクライアントの要求を受けた後に、スレッドまたはプロセスを作成する。この新たに作成された、1 スレッドまたは 1 プロセスのために必要なメモリの量を考える。また、サーバプログラム自身が必要とするメモリの量を考える。

Mach 上でスレッドを使った場合、Mach 上で従来の方式を使った場合、従来の UNIX 上の場合の 3 通りについて測定を行なった。（表 5.1）

Mach 上でスレッドを使用
: スレッドに必要となるメモリの使用量は少ない。ここで使用されるメモリは、スレッド毎に必要とするスタック領域と考えられる。

Mach 上でプロセスを使用
: メモリの使用量はスレッドを使用した場合より大きいが、UNIX の場合に比べれば少ない。これは Mach の仮想記憶機構の効果である。プロセスの複製を生成する場合、プロセスの仮想メモリ空間をすべて共有し、共有された空間に変更が加えられるまでメモリの複製を作らない。このため参照するだけの領域は共有され、変更される領域とスタック領域の分だけ使用される。

UNIX でプロセスを使用
: メモリ使用量はもっとも多い。プロセスの複製が生成される時に、データ領域とスタック領域が確保され、その量だけ必要となる。[1]

スレッドを利用したサーバプログラムは、他の場合と比べてプログラムサイズが大きくなる。これは C-Thread ライブラリが必要とする部分と考えられる。

クライアントの数が少ない場合は大きな効果は得られないが、従来の UNIX と比較を行なえば効果があると考えられる。クライアントの数が多い場合、より少ないメモリでの実行が可能となり、スレッドの使用が効率的であると考えられる。

方式	Thread(Mach)	Process(Mach)	Process(UNIX)
使用量 (Kbyte)	4	8	16

表 5.1: メモリの使用量

[1] 従来の UNIX は LUNA-88K とはアーキテクチャが異なり、正確なデータとは言えない。しかし複数のアーキテクチャで測定を行ない、同様な結果を得ている。

5.4.4 速度

まず、クライアントの要求を受けサーバが新しいプロセスを作成し対応するまでの時間を考える。スレッドを利用する場合と従来のプロセスの場合との違いは、新しいスレッドまたは新しいプロセスの生成に必要な時間の差であると言える。そこで新しいスレッドまたはプロセスを 1 つ生成する際に必要な時間の測定を行なった。(表 5.2)

方式	Thread	Process
時間 (ミリ秒)	3.0	8.0

表 5.2: 1 スレッド /プロセスの生成速度

また複数のスレッドまたはプロセスを並列実行する場合の違いを測定した。(表 5.3) プロセスの場合は実行するプロセスの数に比例した時間を必要とするのに対し、スレッドでは実行する数が増える割に処理時間を多く必要としない。同一タスク内のスレッドではメモリ空間を共有しているため、アドレス空間を切替える時の処理が軽減されるためと考えられる。

並列実行数	1	2	3	5	10	20
Thread	1.5	1.6	1.8	2.9	4.1	6.8
Process	1.8	3.9	6.0	10.4	20.7	41.1

表 5.3: 並列実行時の処理速度

スレッドを利用することでサーバの反応時間の短縮が期待できる。またクライアントの数が多くなるほど処理速度の向上が期待できる。

第 6 章

今後の課題

- 今回作成したプログラムは実験を主目的にしたため、現在計算機上で使われるプログラムと比べ実用性が低い。現在計算機上で利用されているサーバプログラムを変更してスレッドの使用を試み、より多くの方向から検証する必要がある。
- 今回は複数のプロセスを生成する代わりにスレッドを使用した。従来、1 プロセスで全クライアントの要求を処理していたサーバプログラムでも、スレッドを用い処理を分散することで効率的に動作すると考えられるので、その様な場合の実験を行なう。
- プログラム作成の負担に関してさらに考察が必要と考える。
- 本研究では Mach を対象に行なったが、Mach 以外の分散オペレーティングシステムの効果的な利用を考慮する。

第 7 章

結論

本研究では、Mach オペレーティングシステムの持つ、タスクとスレッドの機能を利用することで、サーバプログラムの効率的な実装が出来ることを提案した。そして実際に実験用サーバプログラムの作成と考察を行なった。

その結果、従来のサーバでは複製プロセスを作成していたが、代わりにスレッドを利用することで、メモリ効率や処理速度、情報の管理の面で効率の良い実装が可能であることを確認した。

現在、計算機上で利用されているネットワークアプリケーションに変更を加え、スレッドを利用することで効率の向上が期待できる。しかし現在すでに存在するプログラムの変更を行なう場合には、共有データの排他制御を行なう必要がある。またアプリケーションの種類によってはスレッドの利用が出来ないものもある。