特集：情報プラットフォーム

研究論文

高信頼制御システムのための拡張型分散 OS

Extensible distributed Operating System for reliable control systems

丸山 勝巳
国立情報学研究所
Katsumi MARUYAMA
National Institute of Informatics
児玉 和也
国立情報学研究所
Kazuya KODAMA
National Institute of Informatics
日高 宗一郎
国立情報学研究所
Soichiro HIDAKA
National Institute of Informatics
橋爪 宏達
国立情報学研究所
Hiromichi HASHIZUME
National Institute of Informatics
計 宇生
国立情報学研究所
Yusheng JI
National Institute of Informatics
井手 一郎
国立情報学研究所
Ichiro IDE
National Institute of Informatics
中村 明
国際基督教大学
Akira NAKAMURA
International Christian University

要旨

今やほとんどのシステムがソフトウェア制御になっており，システムの成否をソフトウェア開発が握っているといえる．中でも制御システムのソフトウェア開発は難題である．制御システムには，公衆通信網やプラント制御のような大規模な実時間制御システムから，情報家電のような小規模なシステムまで多種多様ある．制御システムのソフトウェア開発が困難である理由の一つは，適切な OS が無いことにもよる．本稿ではこのような多様な制御システムのための OS が提供すべき諸性質についての検討を行い，最新のマイクロカーネル技術とマルチサーバー方式によりそのような OS が構成出来ることを主張している．検討中の OS は多様な要求に応えられる拡張性，高信頼性，高度な分散処理をサポートする事が期待される．現在 L4 マイクロカーネル上にマルチサーバーOS である Minix の移植を進めながら割り込み処理と保護機能の問題，論理空間切り替えの効率の問題，分散処理のために必要な枠組について検討している．

ABSTRACT

Demands of operating systems for control systems range from real-time systems to intelligent home appliances. However, what is needed depends on their applications. This paper addresses OS characteristics desired to meet above demands and proposes that such OS consists of multi servers running on state-of-the-art μ-kernel. Our OS is expected to support extensibility, reliability, and highly-distributed processing functionalities. Porting of Minix multi-server OS on L4 μ-kernel is underway. Issues such as interrupt handling, protection, switching of address space, efficiency aspects and framework to distributed processing are also described.

[キーワード]

OS，マイクロカーネル，マルチサーバーOS，拡張型 OS，プログラム保護，分散処理，能動オブジェクト

[Keywords]

OS, Micro kernel, Multi-server OS, Extensible OS, Program protection, Distributed processing, Active object

1　はじめに

今やほとんどのシステムがソフトウェア制御になっており，システムの成否をソフトウェア開発が握っているといえる．中でも制御システムのソフトウェア開発は難題である．制御システムには，公衆通信網やプラント制御のような大規模な実時間制御システムから，情報家電のような小規模なシステムまで多種多様ある．制御システムのソフトウェア開発が困難な理由の一つは，適切なOSが無いことによる．

このような制御システムは，適用分野毎に特別な機能や性能が要求される．例えば公衆通信網システムでは，超多重処理・実時間性能・高信頼性，融通性の高い分散処理機能が要求される．また，小型の組込システムでは多機能よりも経済性が要求される．このため，要求条件が厳しいシステムの場合には独自OSを開発し，要求条件が緩い分野ではVxWorks, pSOS, ITRON等の商用OSやモニター的簡易OSが使われている．特に中ないし大規模な制御システムでは，多様な機能，高信頼性，高性能が要求されるので，適切なOSが望まれる．

以上の観点から，我々は以下を特徴とするOSの研究を開始している．

拡張性・適応性 マイクロカーネル技術とマルチサーバーOS技術により，機能拡張や独自機能の追加の容易化，スケジューリングアルゴリズムの変更等を可能とする．

分散処理 NWを介して多数のリソースを効果的に制御できる必要がある．分散オブジェクトの機能をCORBA[1]のようなミドル階層ではなくOS自体が提供することにより効率的で融通性に富んだ分散処理を容易に実現できるようにする

堅牢性 特に組み込みシステム用簡易OSは，プログラム保護機構などがなく，障害によりシステム全体がダウンすることが多い．また，この事は，プログラム開発が困難となる原因でもある．強固なガード機構による信頼性の強化は重要である．

QoS保証 巨大ストリームデータのサービス品質保証を出来る仕組みを提供する．

本稿では，第2節で拡張型分散OSの必要性と要求条件について述べ，それに対する本研究のアプローチについて第3節で述べる．次にその要素であるL4マイクロカーネル，マルチサーバー構成，分散処理機能についてそれぞれ4，5，6節で述べる．

2　拡張型分散 OS の必要性と要求条件

先ず制御システム用OSに望まれる条件を概観してみる．

1．適用分野の要求に応じて，機能の追加・変更が容易に行えること．
2．性能やアルゴリズムのチューニングが可能なこと．例えば，ユーザ独自のスケジューリングアルゴリズムやメモリ管理アルゴリズムを容易に組み込むことが出来る．
3．プログラムプロテクションの強化
現在の小型実時間OSはプログラム保護機構が弱いので，部分プログラムの障害等によりシステム全体がダウンすることが多い．特に，制御システムのプログラムの処理の中核は，

多数あるいは多様なハードウェアリソースの管理・制御であるが，従来のOSの仕組みでは，ハードウェアの制御は一般にカーネルモードで実行させる必要がある．ユーザモードのプログラムはプロテクトされた環境で走り，高度なデバッガーも使えるが，カーネルモードで走るプログラムの開発は，プロテクトも無く高度なデバッガーも使えないため格段にプログラム開発が困難である．そこで，ハードウェアリソースを制御するプログラムも，ユーザタスクと同様にプロテクトされた環境で走れるようにしたい．

4．多重処理
一般に制御システムは並行動作する多数のリソースやサービスを制御する必要があるので，効率的で融通性の高いマルチスレッド機能が望まれる．

5．分散処理機能
ネットワークを介して多数のリソースやサービスを協調動作させることが要求される．また，分散処理形態も Client-Server型分散処理だけでなく，各オブジェクトが対等の立場で通信し合うPeer-to-Peer型分散処理が今後は重要になる．また，メッセージのやりとりも同期型の他に非同期型通信が要求される．さらに，使い易いサービスの登録や検索機能が要求される．

6．ブロードバンド時代のサービスサーバとして，実時間ストリームデータを扱うためのサービス品質保証のための基本機構もOSが提供することが望まれる．

3　本研究のアプローチ

このような要求に応えるために，我々は以下のようなアプローチを採っている．

3.1　マイクロカーネル型 OS

OS機能は大別して，メモリ管理・スレッド制御等の「プリミティブ機能」と，ファイルサービス機能やネットワークサービス機能などの「OSサービス機能」とに分類できる．代表的なOS構造には，プリミティブ機能からOSサービス機能までを一体化したモノリシック型OSと，プリミティブ機能とOSサービス機能を分離して後者をユーザモードで走るタスクとしたマイクロカーネル型OSとがある．高度のハッカーが適切にプログラム開発をするならば，効率的には前者が優れるが[1]，システムのモジュール化，拡張性，プログラム開発の容易化のためにはマイクロカーネル型OSが有利である．

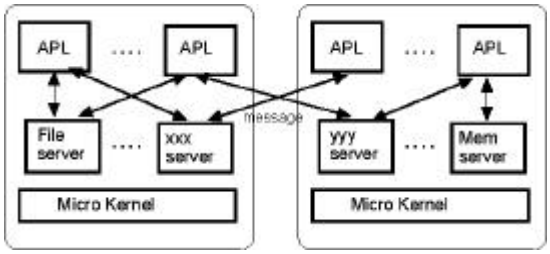

図 1: マルチサーバー構成

今までの制御システムのプログラム開発の困難は，ハードウェア制御のために大部分のプログラムをカーネルモードで実行させる必要があり，このため高度のデバッガーが使えなかったこと，カーネルプログラムの微妙なテクニックが必要であることにある．マイクロカーネルでは，OSサービス機能はユーザモードで実行させることが出来るので，プログラム開発の大幅な簡易化が期待できる．

マイクロカーネル方式が効率的に不利になるのは，システムコールがモノリシックOSではトラップ割り込みによる手続き呼び出しで済むところを，マイクロカーネル方式ではタスク間のメッセージ通信になるからである．このメッセージ通信化によるオーバーヘッドは，優れたマイクロカーネルならば 1回あたり数十命令であり，システムコールが数千命令に1回とすれば，総合オーバーヘッドは数%におさまり，むしろシステム全体の最適化によりトータル性能は向上が期待できる[2]．マイクロカーネルの方が，ユーザ独自のOS機能を追加しやすく，またユーザ独自のスケジュールやメモリ管理のアルゴリズムを追加しやすい．

3.2　マルチサーバーOS

既存のマイクロカーネルを使ったOSでも，OSサービス機能(タスク管理，ファイルサービス，セッションサービス，NWサービス等)全体を単一のタスクとしているものが多い(*Cf.* L4Linux[3], MachLinux等)．この構成では，OSサービス機能をユーザレベルのタスクとして扱える利点はあるが，それ自体一体化した大きなプログラムであるので，保守や機能追加はまだ十分

[1] そのプロジェクトに Linus Torvalds 並のハッカーが永続して居るならば，モノリシック構造を採用するのは良いアイデアである．

に容易ではない．とくに制御システムでは，多種多様なOSサービス機能が要求されるので，個々の機能単位毎（例えばファイルサービス，ネットワークサービス）に独立タスクにすることもできることが望まれる（図1）．

3.3 分散処理の OS サポート

融通性に富む分散処理，リモートリソースもローカルリソースと同様に扱えること等を実現するために，OSとして以下の機能をサポートする．

1. Peer-to-peer分散処理のための機構のサポート

実用化されている分散処理用基盤システム(例：CORBA[1], JAVA-RMI[4])では，クライアントがサーバーに要求を送って返答が返ってくるのを待つというClient-Server型処理を主体にしている．つまり，クライアントとサーバーは非対称な役割分担のもとに逐次的に動作するので，Remote Procedure Callとして実装される[2].

これに対し，高度あるいは大規模な分散処理では，多数の分散オブジェクトが対等の立場で非同期にメッセージを送り，必ずしも返答を待たずに処理を進め，その返答は必要になったときに初めて受け取るような方式も重要となる．丁度，人間社会で多数の人々が電子メールで情報をやりとりしながら仕事を進めるようなものである．このような分散処理モデルを，ここではPeer-to-peer型処理と呼ぶことにする[3].

このために，グローバルな非同期メッセージ機構，つまりリモートアクセスされるスレッドにはグローバルIDを与えて，このグローバルIDを使って目的オブジェクトに非同期メッセージを送れるようにする．また，非同期メッセージに対して，非同期に返答を返せる機構 (Futureメッセージ)も用意する．

この機構により，能動オブジェクトを使った分散処理システム[5]も，容易に実現できる．

2. リモートリソースの操作

大型制御システムでは，非常に多数(10^4個のオーダ)のリソースをローカル・リモートを問わずに効率的かつ簡単に操作することも要求される．このために，リソースサーバは「要求メッセージを受けて目的リソースを制御するmanagerスレッド(能動オブジェクト)」と「個々のリソースを表す受動オブジェクト」からなる構成とする（図2）．前者はグローバルIDを持ちグローバルにメッセージ交信が出来，後者は前者に従属して動作する．プログラマには，リモートオブジェクトは managerスレッドのグローバルIDと個々のリソースを指すローカルIDを含む単一のグローバルオブジェクトIDとして見え，ネームサーバ，サービスブローカ若しくはグローバル名前空間を使ってグローバルオブジェクトIDを得ることにより，自在にリソースを操作できるようになる．

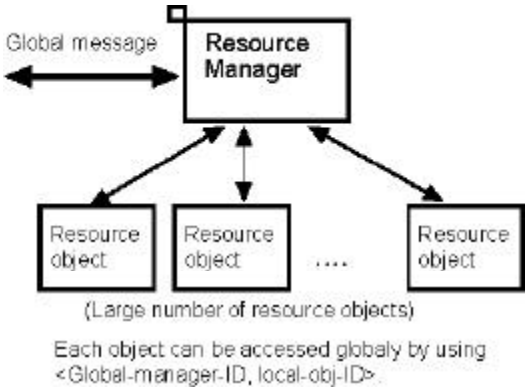

図 2: リソースサーバー

3. サービスの登録・公開・検索

リモートオブジェクトやリモートサービスを簡単に登録でき，簡単に利用できるようにするために，目的オブジェクトを名前から検索するName Server，サービス機能・仕様から検索するService Brokerを作るための機構を提供する．

4. 冗長性の確保のためのオブジェクトの複製と，メッセージのマルチキャスト

大規模システムでは信頼性確保のために，オブジェクトの複製の分散配置，要求メッセージのマルチキャスト，返答メッセージの選択受信の機能を検討する．

3.4 QoS 制御とプロトコルスタック

プロトコル処理プログラムは，巨大ストリームデータのQoS制御機能など色々な機能を容易に組み込めること，高効率であることが要求される．

プロトコル処理プログラムの構成法としては，x-kernel[6]方式が融通性と効率の面で優れている．例えば，ビデオデータストリームのジッターを吸収するプ

---

[2] CORBA や Java-RMI の分散処理は, stub と skeleton という仲介ルーチンを使った Remote Procedure Call として実装されている．なお，CORBA では非同期呼び出しも可能であるが，プログラム記述は難解である．

[3] Internet での Peer-to-peer 処理とは，必ずしも同義ではない．

ログラムを書いて，プロトコルスタックとして導入するような事が容易に行える．x-kernelは，マルチスレッドを前提としており，到着パケット毎にスレッドを割り当てて処理をさせている．我々が採用したL4マイクロカーネルは，次項で述べるように効率の良いスレッドを提供しているので，x-kernel的手法をベースにQoS制御プロトコル実装法も検討しつつある．

4 L4 マイクロカーネル

マイクロカーネルが提供すべき機能は，(1)論理空間の管理，(2)スレッド制御，(3)スレッド間通信，(4)割り込み対処である．独自開発も含めて目的に合うマイクロカーネルの検討をして，ドイツGMDで設計されたL4マイクロカーネル[2]の利用が効果的であると判断した．L4は，既存のマイクロカーネル (Mach[7], V[8], Chorus[9]等)の技術蓄積が活かされており，必要ならば機能追加・変更をすることで，我々の目的を実現できる．L4は，以下の特色を有する．

1. タスク (論理空間)の管理
   L4では，個々の論理空間をタスクと呼んでおり，任意個を動的に生成・消去できる．論理空間と物理ページとのマッピングを行う機構をpagerと呼んでいる．独自のページングアルゴリズムを使いたければ，ユーザがpagerを実装して組み込むことが出来る．

2. マルチスレッド機能
   個々のタスクの中で最大64個のスレッドが使える．スレッドの仕組みは洗練されており，実時間システム向きの効率を提供する．

3. メッセージ通信機能 (スレッド間通信機能)
   マイクロカーネルの性能を一番左右するのが，スレッド間通信の効率であり，L4はこの点で非常に優れている．L4のスレッド間通信機能は，以下の特色を持っている．
   (a) メッセージの宛て先はスレッドであり，ThreadIDにより指定される (*Cf.* メッセージボックス宛のMach方式よりも，本方式の方が分散オブジェクトプログラムが書き易い).
   (b) メッセージは同期型であり，送り元は宛て先が受理するまでブロックされる(*Cf.* Machは非同期型).
   (c) 宛て先スレッドが同一タスク内でも，別タスクでも，同様にメッセージを送れる．
   (d) 以下のように，非常に効果的にメッセージ転送機構を使い分けできる．
   レジスタ転送 32bits×3個 (Ix86プロセッサの場合)のパラメータはレジスタのみで転送されるので，約100マシンサイクルで転送され[10]，手続き呼び出しよりも少し重いだけである(RISCやIA-64[11]プロセッサは多数のレジスタを持つので，更に多数のパラメータをレジスタ経由で転送できる).
   メモリ転送 それ以上の多数パラメータの場合はメモリコピーを伴うが，一時的ページマッピングを利用した非常に効率的な実装である．
   バッファ転送 異なる論理空間の間のバッファ間コピーも，プロセッサのページマッピング機構を巧みに利用することにより，非常に効率的に行える．また，大きなバッファから複数個の小さなバッファへ，あるいはその逆方向へのコピー(scatter and gather)も単一のシステムコールで可能である．
   ページマッピング ページ単位のデータ転送は，該当ページもマッピングだけで行われる．データコピーが不要なので，非常に効率的に行える．Copy-on-write 機能も Pager で実現出来る．

4. 割り込みの扱い
   L4では，割り込みハンドラはスレッドとして実装し，そのスレッドIDをL4に登録する．割り込みが生ずると，L4は割り込みメッセージを割り込みスレッドハンドラに送り，そこで割り込みが処理されることになる．このように割り込みもメッセージとして扱うので，割り込みハンドラをカーネル内でなくユーザモードで動作するタスク内に置くことが出来る．

なお，L4の実装は数種類あるが，2000年10月にフリーソフトとして公開されたC++で実装したIx86用カーネルは[12]，性能と簡明さがよくバランスした実装であり，我々はそれを用いている．

5 マルチサーバーOS 構成

リソース管理機能のプログラム保護とモジュラリティーを強化するために，ファイルサービス，ネットワークサービスといったサービス単位毎に独立タスクとして実装する．この方式に関わる検討内容等を以下に説明する．

1. プログラム保護と効率

個々のサービスが個別タスクとして走行するため，プログラム保護は頑強である．一方，TLBの入れ替え等タスク切り替えに付随するオーバーヘッドが生じるために実装によっては効率が損なわれる．そこで，現在以下の対策を検討している．

Small space機能の活用　個々のサービスを個別タスクとすると，論理空間の切り替えが増え，プロセッサによってはTLBのflushによるオーバーヘッドが懸念される[4]．

L4 マイクロカーネルはこの点も考えられており，OSサービスタスクの様に頻繁に駆動されるタスクは，TLBのflushをせずにタスク切り替えを行えるアドレス空間(Small space)が提供されている[2]．この機能を利用することにより，プロセッサ種別によってはタスク切り替えのオーバーヘッドが削減され，機能を複数のサーバーに分割したオーバーヘッドを軽減することが出来る．

ReadOnly共用空間の利用　例えばタスク管理ブロック (UnixではProcess Control Block)は，複数のサービスタスクから頻繁にアクセスされる（大部分は読み出しデータ）可能性があるが，これをいちいちタスク間通信でデータを取り出すのでは効率が低下する．そこで，タスクサーバーからは Read / writeアクセス可能であるが，他のサービスタスクからはRead Onlyアクセスできる共用メモリ空間を使うことで最適化をはかる．

2. サービスタスクとドライバの実行モード

デバイスのドライバプログラムは，大部分のOSではカーネルモードで実行させる必要があるのでその開発は非常に困難である．

これに対し，L4カーネルではデバイスのドライバプログラムをユーザモードで実行させることが出来，ドライバプログラムの開発も容易化できる．具体的には，個々のIRQごとに，割り込み時のデバイス処理を行うユーザスレッドを用意して，これをL4 カーネルに登録しておく．あるIRQで割り込みが発生すると，L4カーネルはそれを割り込みメッセージに変換して，そのIRQ用に登録されている割り込み処理スレッドにメッセージを送る．こうした割り込み処理スレッドは，ユーザモードで割り込みに対する処置を行う．

同一IRQに連続して割り込みが発生した場合も，割り込みメッセージを割り込みスレッドに渡す際に自動的にシリアライズされるので（つまり，割り込みスレッドは前の割り込みの処理が終了するまでは，次のメッセージを受け付けないので），正常に処理される．

3. I/O空間のアクセス権

サーバータスクは，原則としてユーザモードで実行されるが，ドライバを含む場合にはI/O制御空間（あるいはmapped IO空間）にアクセスする必要がある．このような場合には，サービスタスクには安全性を保ちながらI/O空間のアクセス権を与える必要がある．

この実現法はプロセッサ依存である．例えばIx86の場合には，次の機構を利用する[13]ことで，ユーザタスクから安全にI/O空間のアクセスをすることが出来る．

- TSSのI/O map base addressが指すbit mapによる制御
- EFLAGS registerのI/O privilege fieldによる制御

4. IRQ sharing

IRQ番号は一般には複数のデバイスで共有されるため，デバイスへのアクセスを担う複数のサービスタスク間で共有する必要がある．共有する方法として，割り込みメッセージを

(a) カーネルがマルチキャストする

(b) サービスタスクが転送する

という選択肢が考えられる．何れの場合もハードウェアに割り込み確認を出すタイミングに注意する必要がある．

6　分散処理機能

汎用の分散処理ツール(CORBA[1], Java-RMI[4]等)では，リモートオブジェクトのProxy(Stub)ルーチンをローカルに用意して，クライアントがProxy(Stub)ルーチンを呼ぶとRemote Procedure Callにより宛先ノードのSkeletonルーチン経由で目的オブジェクトが実行される．これに対し，制御システムの分散処理が要求する性能・信頼性・融通性をOSレベルで実現するために，

[4] MIPS の TLB は process-ID も記憶しているので，論理空間が切り替わっても該当 entry を flush するだけで済むが，X86 系では TLB の全 entry の flush が必要である．

リモートスレッドと自在に直接通信できる事が望まれる．このために，L4マイクロカーネルの外部に以下の機能を追加する．

1. グローバルスレッドID (GlobalThID)

分散OSに要求される第一の機能は，メッセージのグローバル配信である．L4マイクロカーネルでは，ノード(ホスト)は任意個のタスク (論理空間) を生成でき，タスクは最大64個のスレッドを生成できる．各スレッドは，<TaskID，LocalID>からなるローカルスレッドIDを持っており，メッセージはローカルスレッドIDで指定されたスレッドに直接送られる(これに対し，Machではメッセージはスレッドではなくメッセージボックス宛に送られる)．

メッセージのグローバル配信を行うには，宛先ノードの識別子を含んだグローバルスレッドID，つまりGlobalThID: <NodeID，<TaskID，LocalID>>を導入し，OSはこの宛先ノード情報をみて広域にメッセージ配送を行う．

この機能は，効率的にはL4 カーネルの中に組み込むのが有利であるが，効率よりも融通性を重視してカーネル外の分散処理サーバーとして実現する．

2. 非同期メッセージ転送と非同期返答の機構

L4 マイクロカーネルのメッセージは同期型，つまりメッセージの送り手は，受け手がメッセージを受け取るまで待ち合わせる．しかるに，前述の通り我々の分散処理機能では非同期型メッセージも必要で，このためにはどこかでメッセージのバッファリングを行う必要がある．この機能も，カーネル外の分散処理サーバーの中で実現する．

3. ノード間のコネクション

通信でのメッセージ欠落を避けるために，Connection型の高信頼プロトコルの使用と，また障害が生じた時はExceptionを通知する機構を検討している．メッセージはスレッド間で交信されるが，Connectionは，交信しあうスレッド間ではなく，通信しあうノード間に張ることにより，静的かつ動的に効率を上げている（図3）．プログラムでは，分散処理の開始時に相手ノードをOSに教え込むだけで，あとはローカルスレッドにもリモートスレッドにも同様にメッセージを送ることができる．

なお，現在の試作ではノード間通信には(Stream転送向きの) TCPを用いているが，本来はメッセージ通信に適合し，かつ信頼性の確保された(メッセージが確実に送り届けられる)プロトコルが望ましい．

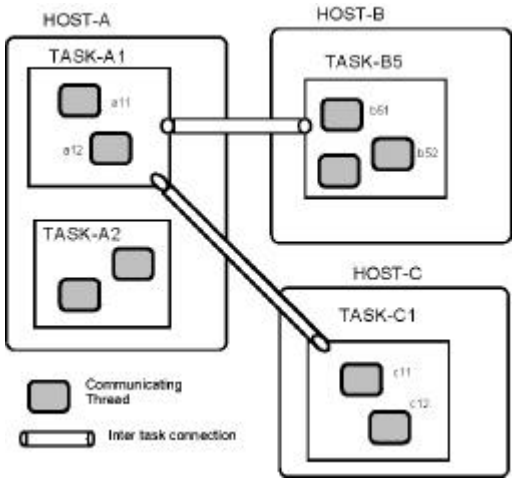

図 3: ノード間のコネクション

4. ネームサーバ・トレーダ・リモートマウント

リモートアクセスのためのサービスやリソースの登録と検索の機構は，普通のサービスタスクとして簡単に実装できるので，OS組み込みサービスとしては特に実装する必要はない．

本OSのサービスサーバは，分散サーバーとして実装するので，リモートリソースもローカルリソースと同様にアクセスできる．なお，応用プログラムにリンクされるライブラリにて，(ファイル入出力の際のバッファーキャッシュ等を対象とした)ローカルCache機能を提供することも検討している．

7 実験システムの実装概要

我々はKarlsruhe大学で実装された L4-KAマイクロカーネルを用いてマルチサーバーOS構築を進めている．

プログラム開発もOS走行テストも同一のPCの上で行っている．つまり，プログラム開発はLinux-PC上のGCCコンパイラで行っている．また，こうして作られた実験OSの走行テストは，同じLinux-PC上にインストールした仮想マシンエミュレータVMware[14]で行っている．実マシン上で走行テストするのに比して，VMware上での走行テストは大変簡便で効果的である．

7.1 マルチサーバー構造 OS

実験OSの概略を図4に示す．タスクサーバー，ファイルサーバー，NWサーバーなどがユーザモードで走るマルチサーバー構造である．

ファイルサーバーとHDDドライバは，Minix から移植した．また，Ether ドライバとインターネットプロトコルスタックの移植も進めている．Minixのファイルサーバーは独立のタスクであるので，この移植は Linux などモノリシックOSからの移植に比して格段に容易であった．

7.2 デバイスドライバとユーザレベルタスク

デバイスドライバプログラムをユーザレベルタスクに置けるのが本システムの特徴の一つであるが，これをどのタスクに置くかに関しては次の3手法が考えられる．

1. 各ドライバ毎に個別タスク化する．
2. ドライバは，関連するサービスタスクに含ませる (Ex. HDD-driverはFile-serverタスクに含ませる).
3. 全ドライバを単一タスクにまとめる．

現在は2.の手法で実装しているが，1.，2.の手法も実装してみて，効率と拡張性の評価をする予定である．各機能間はメッセージ結合であるので，これらの変更は比較的容易に行える．

7.3 別論理空間の間のブロックデータ転送

モノリシックOSでは，ファイルサービスはカーネル内で実現されていおり，ファイルの書き込みや読み出し時のブロックデータ転送は，すべてカーネルが行う．カーネルは，カーネル空間もユーザ空間も自在にアクセス出来るので，最小のオーバーヘッドでデータコピーを行うことが出来る．

それに対し，マイクロカーネル＋マルチサーバー構成のOSでは，ファイルサービスを行うファイルサーバーはユーザレベルのタスクであるので，自前ではユーザ空間との間で直接データコピーすることが出来ず，カーネルに依頼ことになる．従って，このオーバーヘッドを最小化する事が肝要である．

タスク（論理空間）の間で大きなデータをやりとりするシステムコールの動作例を図5に示す．応用プログラム (APL)がファイル読み出し read(fs, buf, sz) を実行すると，ライブラリはこれをファイルシステムへのメッセージに変換する．

ここの説明では，サイズszは2.5キロバイトとする．また，ファイルシステムの個々のバッファーキャッシュは1キロバイトとする．ファイルシステムは，要求データをバッファーキャッシュから検索する．もし，バッファーキャッシュ上に見つからなければ，ハードディスクを駆動して目的データをバッファーキャッシュ上に読み出す．要求データサイズは2.5KBと仮定しているので，3個のバッファーキャッシュ(1KB x 3)の上に目的データが載っている．L4では，3個のバッファーキャッシュから1個のユーザバッファに1回のメッセージ呼び出しでデータ転送する．L4メッセージは，このように分割されたデータでも1回のメッセージで

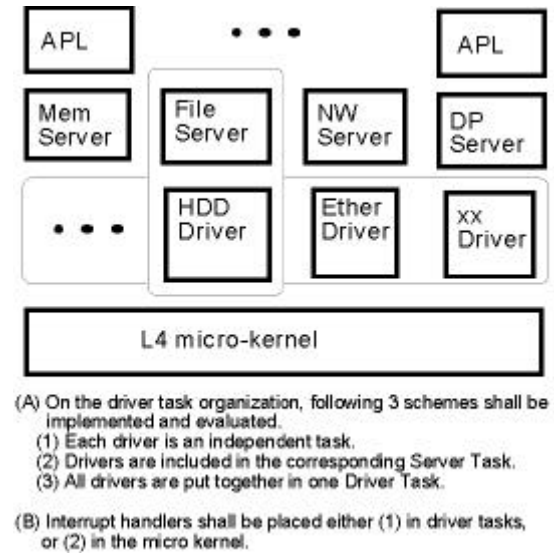

図 4: 試作 OS のタスク構成

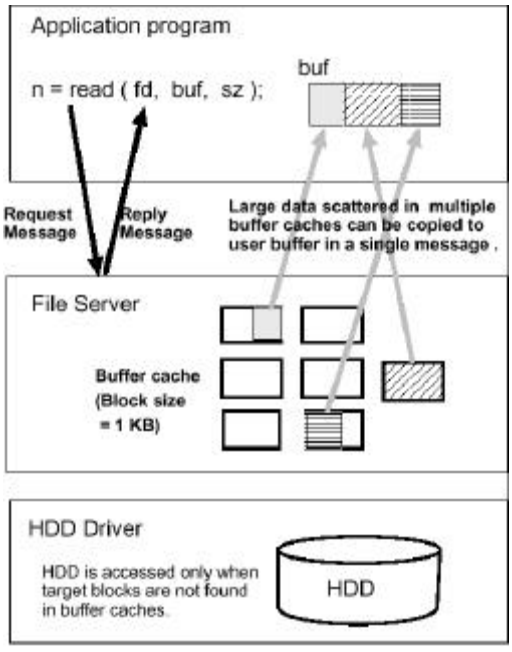

図 5: ファイルシステムとメッセージ

送れる(Scatte and gather 機構)ので，ブロックデータも効率がよく転送出来る．現在，正確な定量評価を行うための準備を進めている．

8 関連研究

Minix[15]は，教育用に簡略化されたマルチサーバー型OSであり，OSサービス機能はファイルサーバー，メモリ管理サーバー，NWサーバーの各タスクにより提供されている(図6)．メモリ管理にはpagingを使っておらず，各タスクの論理アドレスは並行移動するだけで物理アドレスになる．デバイスドライバはカーネルの中に組み込まれており，例えばデバイスとユーザ空間の間でデータをコピーする場合は，ユーザ空間の論理アドレスに足し算をして物理アドレスに変換して，物理アドレスを使ってデータコピーを行っている．このように，大型システムへの発展性には欠ける．

Dresden工科大学では，L4'Linux[16]を実装している．これは，L4マイクロカーネル上の一つのタスクを仮想マシンとして扱い，その仮想マシンの上でモノリシック構造OSの Linux が走るものである．L4の上でLinuxプログラムを走らせるには便利であるが，Linux自体は大きなモノリシック構造なので，OSのモジュール化は達成していない．

IBMでも，L4マイクロカーネルを使ったマルチサーバーOS “ Sawmil ” [17]の研究が進められている．目的はLinuxのAPIをマルチサーバーで提供することであり，現在はLinuxのソースプログラムからファイルサーバーを切り出して，L4の上に実装している．

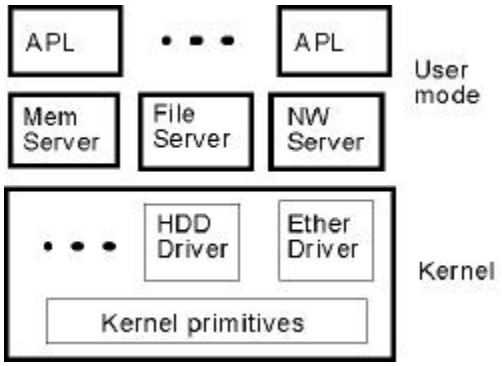

図 6: Minix のタスク構造

9 おわりに

適用分野の要求に応じて，容易に機能追加やアルゴリズムの整合が可能な実時間制御システムOSの構成法について試作を伴う検討を行い，最新のマイクロカーネル技術とマルチサーバー方式により，性能のオーバーヘッドを押さえた上で拡張性，高信頼性，高度な分散処理をサポートする事が可能であることを明らかにした．

例えば，システムコールがモノリシック構造OSではトラップ＋プロシージャー呼び出しですむ所を，マイクロカーネル＋マルチサーバー方式OSではメッセージ通信になり，オーバーヘッドが懸念されるが，最新のマイクロカーネルのメッセージ通信は効率的であることと，モジュール化の向上によりシステムトータルとしての最適設計が行いやすくなるし，プログラム開発も容易化される．

現在L4マイクロカーネル上にMinixのファイルシステムとネットワークシステムの移植を進めながら，具体的な検討を進めている．モノリシック型OSのファイルシステムはカーネル内で動作するので，カーネル空間もユーザ空間も自在にアクセス出来る（その分，危険が伴う）のに対し，マイクロカーネル＋マルチサーバー方式OSでは，個々のサービスタスクは独立の論理空間＋ユーザモードで動作するので安全ではあるが，オーバーヘッドを削減する適切な設計が要求される．更に，分散処理のために必要な枠組についても検討を行った．

文献

[1] Object Management Group, "The Common Object Request Broker: Architecture and Specification", July 1996.

[2] Liedtke, Jochen, "On μ-Kernel Construction", *Operating Systems Review*, Vol. 29, No. 5, pp. 237-250, December 1995.

[3] Härtig, Hermann; Hohmuth, Michael; Liedtke, Jochen; Schönberg, Sebastian; Wolter, Jean, "The performance of μ-Kernel-based systems", *Proceedings of the 16th Symposium on Operating Systems Principles ({SOSP}-97)*, Vol. 31, 5 of *Operating Systems Review*, October 5-8 1997, New York, ACM Press, pp. 66-77.

[4] Sun Microsystems, "Java remote method invocation specification", *Technical report*, 1997, http://www.javasoft.com/products/jdk/1.1/docs/guide/rmi/spec/rmiTOC.doc.html.

[5] Maruyama, K., "Concurrent object-oriented programming for Realtime systems", *Information Sciences*, Vol. 93, No.1-1, 1996.

[6] Hutchinson, Norman C.; Peterson, Larry L., "The x-Kernel: An Architecture for Implementing Network Protocols", *IEEE Transactions on Software Engineering*, Vol. 17, No. 1, pp. 64-76, January 1991.

[7] Accetta, M.; Baron, R.; Golub, D.; Rashid, R.; Tevanian, A.; Young, M., "Mach: A new kernel foundation for UNIX development", *Proceedings of Summer 1986 USENIX Conference*, July 1986.

[8] Cheriton, David R., "The V Distributed System", *Communications of the ACM*, Vol. 31, No. 3, March 1988.

[9] Rozier, M.; Abrossimov, V.; Armand, F.; Boule, I.; Gien, M.; Guillemont, M.; Herrmann, F.; Kaiser, C.; Langlois, S.; Léoñard, P.; Neuhauser, W., "CHORUS distributed operating system", *Computing Systems*, Vol. 1, No. 4, pp. 305-370, Fall 1988.

[10] Liedtke, Jochen, "Improving IPC by kernel design", *Proceedings of the 14th Symposium on Operating Systems Principles (14th SOSP'93), Operating Systems Review*, Asheville, NC, December 1993, pp. 175-188, Published as Proceedings of the 14th Symposium on Operating Systems Principles (14th SOSP'93), Operating Systems Review, volume 27, number 5.

[11] Intel Corporation, "Intel IA-64 Architecture Software Developer's Manual", Santa Clara, CA, USA, Intel Corporation, January 2000.

[12] Liedtke, Jochen, "L4 Nucleus Version X Reference Manual x86", Universität Karlsruhe, http://l4ka.org/projects/hazelnut/docs.asp.

[13] Intel, "Intel Architecture Software Developer's Manual", ftp://download.intel.com/design/PentiumII/manuals/24319102.PDF.

[14] VMware inc., "Secure transportable Virtual Machinees", Home page, http://www.vmware.com/.

[15] Tanenbaum, Andrew S.; Woodhull, Albert S., "Operating Systems: Design and Implementation", second edition, Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1997, Includes CD-ROM.

[16] TU Dresden, L4-Linux Home page, http://os.inf.tu-dresden.de/L4/LinuxOnL4/.

[17] IBM Watson Research Center, SawMill Home page, http://www.research.ibm.com/sawmill/.