

分散共有メモリ型並列計算機の OS 開発のための クロス開発環境

村山正之¹ 斎藤信男²

慶應義塾大学¹理工学部 ²環境情報学部

分散共有メモリ型並列計算機のそれぞれのクラスタ上で動作するマイクロカーネルを、UNIX の 1 つプロセスとして動作させることによって、各マイクロカーネルが協調して動作する OS 開発環境を単一の UNIX 上に実現する試みについて述べる。

このときの最大の問題点とは、UNIX 上の開発環境で動作するマイクロカーネルと、ハードウェア上で直接動作するものとの、機能・ソースコード・性能の違いである。これについて検討し、開発環境の実現方法の提案を行なう。今回、実際に mach 3.0 マイクロカーネルを UNIX のプロセスとして動作させ、満足のいく結果を得た。

A developing environment for an operating system of a distributed shared memory parallel computer

Masayuki Murayama¹ Nobuo Saito²

Keio university

¹ faculty of science and technology

² faculty of environmental information

This paper describes that an attempt to make an environment that micro kernels on each cluster of a distributed shared memory computer work collaborately on an unix computer.

In this case, issues to be solved are what are different between micro kernel under the environment and one under the real hardware from view points of functionality, source code compatibility and performance. We discusses them and successfully ported mach3.0 micro kernel as a process of unix to show this environment is practical.

1 はじめに

文部省重点領域研究「超並列原理に基づく情報処理基本体系」の研究の一環として、我々はCOS(Collaborative Operating System)と呼ぶ超並列計算機用のOSの研究を行なっている。

超並列計算機のハードウェア(JUMP-1)[1][2]についても平行して研究を進めている。効率の良いソフトウェア開発を行なうためにも、開発中のOSに適した開発環境が必要である。本研究はこれに関するものである。

我々は、最初から何らかのハードウェア上に直接OSを載せて開発をするのではなく、まず既存のOS(UNIX)のタスク(プロセス)として超並列計算機用のOSを実現することを考えた。このとき、最大の問題点は、UNIXの上に実現したOS開発環境上で、実用に耐えうるOSを動作させることができるのか、またハードウェア(JUMP-1)上で直接動作させる場合に比べて、どの程度の機能・ソースコード・性能の違いがあるのか、ということである。ここではこの問題点について検討し、これに対する解決方法の提案を行なう。そしてこの評価のために、COSの代わりにmach 3.0マイクロカーネルをUNIXの1プロセスとして動かすことを試み、満足のいく評価結果を得た。

2 COS 開発環境

2.1 JUMP-1 のハードウェアの特徴

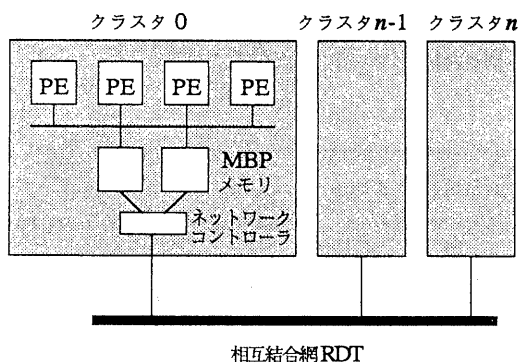


図1: JUMP-1のハードウェアの概要

JUMP-1のアーキテクチャの特徴について説明する。JUMP-1は分散共有メモリ型超並列計算機であり、256個の計算ノードが相互結合網で接続される予定である。各クラスタは、4つの Processing Element (RISC プロセッサ SuperSPARC) とローカルメモリ (16-64MB 程度) からなる共有メモリ型計算機の構成をしている。(図1に、JUMP-1のハードウェア構成を模式化したものを示す。)

各クラスタ間の通信は分散共有メモリ機能を基本としている。アクセスレイテンシの違いこそあるが、各PEは、load/store 命令により、同一クラスタ内のメモリアクセスと、リモートクラスタへのメモリアクセスを同様に扱える。この機能を実現するために、各クラスタのローカルメモリにはMBP (Memory based processor) と呼ぶプロセッサが付随しており、これによりメモリアクセスが管理される。

MBPはクラスタ内のシステムバスに直結されており、PEから物理メモリへのアクセスは、MBPが管理対象とする物理アドレス範囲へのload/storeとして実現される。

MBPは、主記憶上のMBP用のページテーブルを参照することにより、アクセスされた番地を解釈して、ローカルメモリ・または他のクラスタのメモリへのアクセスをおこなう。MBPには相互結合網のコントローラが直結されており、これを介して他のクラスタへのメモリアクセス要求パケットを発行すると共に、他のクラスタからのメモリアクセス要求を受け付ける。また、ローカルメモリをリモートクラスタのメモリのキャッシュとして使用する機能もMBPにより実現される。

このほかにもMBPには、PEからの特定の物理アドレスのアクセスにより、あらかじめ登録しておいたMBPプログラムを起動し、その結果を当該メモリアクセスサイクル中に返す機能などがある。これにより、メモリベースFIFO、メモリベース同期などの機能が、陽な排他制御やシステムコールを発行することなく、論理空間に対する単なるload/storeにより可能となる。マルチタスク環境での、超並列計算機の性能向上に大きく貢献するものと予想される。

2.2 COSの構成の概要

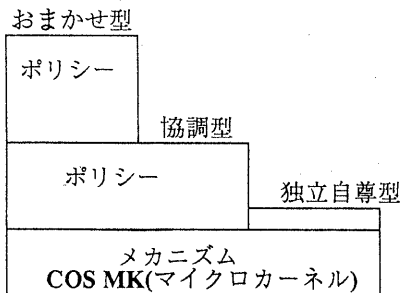


図 2: COS のサービスの階層

COS では JUMP-1 を制御するために 256 個のそれぞれのクラスタ上で自律性をもった仮想記憶方式のマルチタスク OS を動かし、これらが協調的に動作することにより、システム全体を制御する方式をとる。

COS は、超並列計算に対する各種ユーザからの要求を採り入れて、以下に挙げる 3 種類のユーザクラスをサポートする。

● おまかせ型

COS が既存機能として提供する超並列 UNIX 環境で、ユーザプログラムが動作する。物理メモリ、ディスク、ファイルシステム、ページング、スワッピング、タスクスケジューリングなどのシステムリソース管理は、COS が提供するマルチユーザ・マルチタスクのポリシーを使用する。

● 協調型

COS の既存の超並列 UNIX 環境は使用しないが、リソース管理のポリシーは、COS のマルチユーザ・マルチタスク環境のポリシーを使用する。

● 独立自尊型

リソース管理の多くをユーザの責任のもとで行なう。

これを実現するために、COS はメカニズムとポリシーの分離を OS の構造に反映させた設計になっ

ている。つまり、COS の最下層にはメカニズムの実現を担当するマイクロカーネルがあり、その上に各種ポリシーの実現を担当する各種のサーバを置く。COS のサービスレベルの階層を図 2 に示す。

COS のマイクロカーネル上で動作する各種のサーバについては、既存の mach などのマイクロカーネルの構造をもったオペレーティングシステム上で開発が可能であろう。問題は、COS のマイクロカーネル部分の開発である。

この部分は JUMP-1 のハードウェア (CPU、MMU、I/O、分散共有メモリ機能) を直接操作し制御するために、マイクロカーネルの開発環境に対してハードウェアエミュレーション機能を要求するが、これが開発環境の実現を難しいものになっている。

2.3 開発環境への要求項目

OS 開発環境として一般に重要と考えられる項目を以下に挙げる。

● 開発ツールの充実

COS は、C 言語で記述する予定であり、開発環境の OS で UNIX のプログラミング環境が使用できることが望ましい。

● ターゲットマシンとのソース互換性

ターゲットマシン動作後にもソースコードが一貫することが望ましい。

● ターゲットマシンとのバイナリ互換性

JUMP-1 が SuperSPARC を採用する予定なので、開発環境も SPARC アーキテクチャのプロセッサを使用していることが望ましい。

これに加えて、前述した分散共有メモリ型超並列計算機 JUMP-1 のアーキテクチャの特徴に起因する COS の特徴、つまり、各クラスタ上でマイクロカーネルが協調して動作し、これらが相互に各クラスタのメモリをアクセスするために、以下の機能も必須であると考える。

● 多数のクラスタのエミュレーションの実現

マルチスレッドを効率的に動かせるハードウェア、もしくは多くのタスクを効率的に動かせることが望ましい。

- 分散共有メモリのエミュレーション機能

MBP の動作をエミュレートする機能を簡単に組み込めることが望ましい。

2.4 開発環境の実現方法

OS などのハードウェア機能を直接操作するソフトウェアの開発では、いくつかの開発方法が考えられる。

1. 実機にてインサーキットエミュレータなどの開発支援機器を使用
2. ハードウェアのシミュレータの利用
3. ハードウェア操作を開発環境上でそれと等価な機能をあたえるサブルーチンに置き換え、通常の OS 上でアプリケーションプログラムとして開発

1. は、開発後期に発生する解決が難しいバグに対処する場合に、ぜひ必要な方法であるが、開発の当初から必要ではない。また、我々のスケジュールでは、ハードウェアも OS と同時に研究開発をすすめるので、現実的でない。

2. は、ソフトウェアにて仮想的にハードウェアを実現する方法のなかでは、ターゲットとなるハードウェアの機能を一番忠実にエミュレートできる可能性を持っている。しかし、MMU (メモリ管理ユニット) の機能をエミュレートするには、PE が実行する機械語のインタプリタを実現して、毎回のメモリフェッチに対してアドレス変換の計算とメモリ保護のチェックを行なう必要がある。このためシミュレータ上でのプログラムの実行が極めて低速 (実機で動かす場合に比べて 1/1000 程度) になることが予想される。

そこで、COS のマイクロカーネル部の開発環境としては、3. の方式つまり、既存の OS のアプリケーションプログラムとして複数の COS のマイクロカーネルを協調して動作させることを基本方針とした。

現在は、Mach 等のマイクロカーネル上に、各種の OS のインタフェースをサーバとして動作させることが盛んに行なわれている。しかし、マイクロカーネルは OS サーバに対して、ハードウェア独立な仮想メモリの管理機能を提供するため、今回の

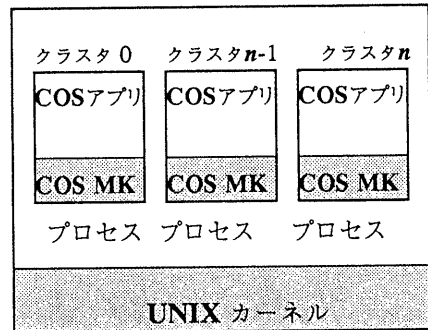


図 3: 開発環境中での COS の構成

ように、マイクロカーネル自体の開発に使用するには、相入れない性質をもっている。

また OS サーバは、そのマイクロカーネルが規定する枠組みに沿ってプログラムを記述する必要があり、開発環境上で動作したマイクロカーネルをターゲットマシンに移植する際にソースコードの追加変更が予想される。

したがって、IBM 社の VM (Virtual Machine)[3][5] にみられるような、既存の OS 上でも JUMP-1 上でも、同一のソースコード、できれば同一のバイナリが動作することが理想であり、これを実現したいと考えた。使用する OS としては、もっとも手に入りやすく、プログラミング環境として優れている UNIX を選び、図 3 に示すように、単一の UNIX のもとで、複数の COS マイクロカーネルのプロセスを動作させることにより、COS 全体の動きを UNIX のなかで実現する。

UNIX の 1 プロセスとして COS を動かせる場合の長所短所について以下に述べる。

2.5 長所

- ソフトウェア開発環境が優れている

現在 UNIX で実現されている豊富なソフトウェアが使用できる。

- デバッグのための情報収集が容易

開発環境上の COS がダウンしても開発に使用している計算機システム自体がダウンするわけではないので、COS のシステムダウン時に

原因解析のための情報を取り出すのが容易である。

- 入出力等周辺装置のドライバを書く必要がない
開発環境である UNIX の read/write システムコールを利用する。これが非同期 I/O をサポートすれば、よりきめの細かい I/O 動作のエミュレーションが可能になる。

- マルチプロセッサのエミュレーションが可能
COS マイクロカーネルを実行するプロセス内部に複数のスレッドを作るなどの手法により、マルチプロセッサをエミュレートすることができる。

- 複数クラスタ間の共有メモリをエミュレート可能

複数の COS プロセスを同時に生成して、この間で UNIX が提供する共有メモリ機能やファイルマッピング機能により分散共有メモリによるクラスタ間の通信をエミュレートできる。

2.6 短所

- 性能測定にはむかない

UNIX のシステムコール組み合わせることにより必要なハードウェアの機能をソフトウェアにてエミュレートすることになるため、開発環境上の COS のオーバヘッドには UNIX のオーバヘッドも含まれる。このため性能測定には向かないと思われる。

- 分散共有メモリの各種コンシステンシモデルを忠実に模倣できない

キャッシュやバスの動作をソフトウェアにて忠実に模倣することは、極めてコストが高く、動作が低速になる。しかし COS が動作するためには JUMP-1 の分散共有メモリ機能がサポートするキャッシュコンシステンシモデルすべてが必要というわけではない。弱いコンシステンシは、主に性能向上のために考え出されたものであり、完全なコンシステンシがあれば一通りの機能は実現できる。したがって、分散共有メモリ機能を UNIX の共有メモリ機能

(完全なコンシステンシ) で代用するにとどめることが可能である。

3 技術的検討

3.1 視点

本節では、COS のマイクロカーネル部の様な仮想記憶方式の OS のソースコードを UNIX 上のプロセスとして動作させることの可能性について検討する。

ソフトウェアとしての OS の特殊性は、OS が動作するために幾つかのハードウェア機能の存在を前提条件にして、これを直接制御していることである。したがって、このハードウェア機能を明らかにして、これに対応する UNIX の機能を調べる必要がある。

そして、この機能を置き換える事により、UNIX のプロセスとして動作させた場合の、仕様の相違(機能制限)、ソースコード変更の度合い、性能の違い、の視点から検討を行なう。

なお、プロセスのアドレス空間のデザインについては、一つのアドレス空間のなかにユーザ領域とシステム領域とを同居させる。システム領域は全てのプロセス間で同じアドレスの範囲にあり、共有されることを前提条件にする。なぜなら、最近のマイクロプロセッサ(とくに RISC)の MMU は、このようなアドレス空間モデルをサポートする機能をもっているため¹、OS の実装が自然になるからである。

3.2 検討すべきハードウェア機能の抽出

仮想記憶方式の OS を実現するには、アドレス変換機能やメモリ保護機能をもった MMU が必要である。CPU は、アクセス違反などにより MMU から通知されたページフォルトをトラップとして受け付け、後に命令の実行を再開できることが必要である。また、OS が動作するにはインターバルクロック、ディスク、コンソールなども必要である。

¹SPARC reference MMU や MIPS R3000 の MMU には、TLB や論理アドレスキャッシュのタグ比較の際に、システム領域についてはアドレス空間 ID を無視することを意図した機能が盛り込まれている。

これらをまとめるとOSが必要とするハードウェア機能は以下になる。

- **トラップ** ページフォルト、割り込み、インターバルクロック
- **MMU 関連** 論理空間、物理メモリ、アドレス変換機能、メモリ保護機能
- **デバイス**
ディスク、コンソール

この他に、スタック管理なども必要であるが、現在の多くのマイクロプロセッサでは、スタックに対する操作はユーザモードで行なうことができるので、開発環境が特にサポートすべき機能として挙げる必要はない。

3.3 トラップ

ページフォルト、インターバルクロック、I/Oの状態の通知は、UNIXでは、すべてシグナル機能で表わされる。

調査したUNIX(SunOS R4.1.x)では、シグナルハンドラからリターンすると、シグナルを受け付ける直前のレジスタ状態に戻り、正しく命令を再開することができる。したがって割り込みやページフォルトの置き換えとして、シグナルを使用することができる。

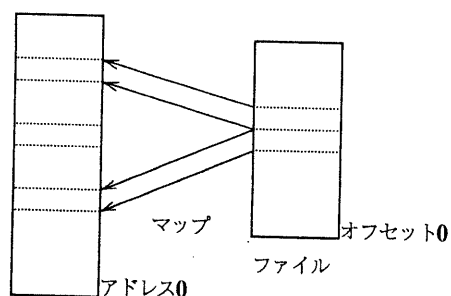
表1にシグナルとの対応関係の表を挙げる。

3.4 MMUのエミュレーション

UNIXには、ファイルを論理アドレス空間にマップして使用する1レベルストレージ機能(mmapシステムコール、図4参照)を実現しているものが多い。MMUが提供する機能には、アドレス変換機能・メモリ保護機能があるが、これの機能の置き換

ハードウェア事象	シグナルの種類
ページフォルト	アクセス違反 (sigsegv)
メモリアライメント違反	バスエラー (sigbus)
システムコールトラップ	特権命令違反 (sigill)
割り込み	I/O 可能通知 (sigio)
インターバルクロック	タイマー (sigvtalrm)

表 1: トラップ・割り込み等とシグナルとの対応



UNIXプロセス
の論理空間

図 4: UNIX(SunOS)での mmap 機能

えには mmap システムコールが有望である。この場合には物理メモリはファイルとして実現される。

mmap システムコールの仕様では、ファイルを論理アドレス空間にマップする際に、ページ単位でマップするアドレスと read/write/execute の組み合わせによる保護条件を設定できる。保護条件に違反したアクセスを行なった場合には、前述のシグナルが発生する。これはページフォルトに相当する機能である。

これらのことより、MMUが提供する機能を十分に満たしているので、MMU機能のエミュレーションを mmap システムコールで行なうことが可能である。

開発環境上の COS マイクロカーネルの論理アドレス空間の大きさは、UNIXがユーザプロセスに許す最大の大きさに制限される。多くのUNIXの実装では、UNIXのカーネル自体が32bitアドレス空間の一部を占めているので、COS マイクロカーネルが32bitアドレス空間の全てを利用することはできない。しかし、OSの基本的な動作の検証を目的とした開発環境では、扱えるアドレス空間の大きさを多少制限しても実質的な問題はない。

4 デバイスの扱い

ディスクとコンソールについては、UNIXのシステムコールにより open, read, write, close 等を実現すればよい。

5 試作結果の評価

UNIX の COS マイクロカーネル部の開発環境としての可能性を検証し評価するために、COS の代わりとして、Mach3.0 のマイクロカーネル部分を SunOS R4.1 (SPARC プロセッサ用の UNIX) 上の 1 プロセスとして動作させることを試み、これを達成することができた。Mach を試金石として選定した理由を以下に挙げる。

- COS と同様にマイクロカーネルである
- 実戦的なマルチタスク OS である
- 仮想記憶方式でありページフォルトなど MMU の機能を十分に活用している
- ソースコードが公開されている

なお、UNIX 上での開発環境で動く mach マイクロカーネル (ここでは便宜上 ux-mach と呼ぶ) と、ターゲットである SuperSPARC 上で動作するもの (単に mach と呼ぶ) との比較をしなければならないが、後者はまだ存在していないため、今回の変更作業 (mips 版を sparc 用に変更) の経験をもとに推察を含めて評価する。

5.1 仕様の相違の評価

mach 上で動作するタスクへの影響 (mach3.0 の仕様変更) の見地から比較する。

- システムコールインタフェース

仕様変更はない。

- 論理空間の大きさ

mach マイクロカーネルの論理空間デザインは図 5 の様にした。扱えるタスクのアドレス空間の大きさは 512MB に制限されている。

- MMU 関連

ページフォルトを発生させる仕掛けは実機上と同様に実現できた。mach の論理アドレス管理部については変更の必要がなく、したがってページングについては、実機と全く同様に behaving している。

■ UNIX のシステム空間
□ UNIX のユーザ空間
UNIX プロセスの論理空間

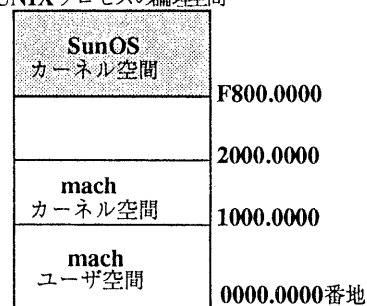


図 5: ux-mach での論理空間の割り付け

問題点として残ったのは、mach ユーザ領域と mach カーネル領域が一つの UNIX プロセスのユーザプログラム領域に同居しているため、mach カーネル領域を mach ユーザプログラムの不当なアクセスから保護することができないことである。これは、システムの信頼性に関することであり、我々が必要としている COS の開発環境では、OS の基本的な動作を検証することが目的であるので、大きな問題ではないと考える。

ユーザ空間内、ユーザ空間同士のメモリ保護については、全く同一の機能を提供している。

5.2 ソースコード変更の評価

5.2.1 mach のハードウェア独立部分

変更なし。全体で約 22700 行 (mig で自動生成された部分は含まず)。Mach ユーザプログラムが利用するシステムコールライブラリ、cthread ライブラリも無変更で使用できる。

5.2.2 ハードウェア依存部分 (アセンブラで記述)

総量で約 500 行ほどある。今回は mips 版の mach のアセンブラ部分をほぼ機械的に sparc アセンブラに書き直した。

この部分の大半は、ターゲットマシンでもほぼそのまま使用できる。変更の対象となるのは、トラップまたは割り込み発生時の前処理と後処理の部分である。この部分には、原因の解析、トラップの原因となった参照アドレスの読みだし、プロセッサの状態の退避、カーネルスタックへの切り替えなど

が含まれている。この作業を行なうために、MMU のレジスタなどを直接読み出したり、また mach と ux-mach ではトラップ発生時の SPARC のレジスタウィンドウの状態が異なるため、処理の手順変更する必要があるだろう。

ほかに sparc 特有のレジスタウィンドウ関連のトラップ (window overflow と window underflow) を扱う部分が必要となる。ux-mach では、この部分は SunOS が行なうので不要であった。

以上の変更をまとめると、ステップ数にして約 1000 行の追加変更が必要であろう。

5.2.3 ハードウェア依存部分 (C で記述)

総数で約 2200 行程度ある。トラップ発生 (シグナル) 時のレジスタの退避方法、およびスタックの管理構造体が unix 上で動作させる場合と直接ハードウェア上で動作させる場合とで異なる。約 500 行ほど変更が必要であろう。

5.2.4 変更評価のまとめ

以上より mach と ux-mach (総行数約 25400 行、デバイスドライバは除く) では、仕様上はほぼ同一であり、ソースコードの違いも約 1500 行、約 6% であろうということが推察される。また、mach のハードウェア独立部には変更の必要がなかった。

したがって、UNIX は、仮想記憶方式の OS の開発環境として十分な機能を持っており、我々の提案する方法が実現可能であることがわかった。

6 おわりに

本稿では、UNIX 上での分散共有メモリ型超並列計算機の OS の開発環境を提案した。これを実現する際の問題点である仮想記憶方式のマルチタスク OS を UNIX 上で動作させる方法を示し、mach マイクロカーネルを用いて実際に評価を行い、十分実用に耐えるものであるという結果を得た。

参考文献

[1] 文部省重点領域研究「超並列原理に基づく情報処理基本体系」第 2 回シンポジウム予稿集,1993.9

[2] 文部省重点領域研究「超並列原理に基づく情報処理基本体系」第 3 回シンポジウム予稿集,1993.9

[3] Deitel,editor "An Introduction to Operating Systems" pp601-629, Addison-Wesley, 1984

[4] "SunOS R4.1 Reference Manual" Sun Microsystems, Inc, 1990

[5] S.E.Madnick;John J. Domovan: "Operating Systems", MacGraw-Hill, Inc.,1974 (池田克夫訳:「オペレーティングシステム」、日本コンピュータ協会、1976)

[6] "MACH Kernel Interface Manual" CMU , Feb.1988