オペレーティング・システム　48－8
（1990．　9．　7）

分散OS ToM におけるファイル・システム

立木秀樹　慶應義塾大学環境情報学部
服部隆志　京都大学数理解析研究所
針間正紀　日本・データゼネラル

ToM ファイル・システムの概要を説明する。マシンごとのローカルな名前と、ユーザごとに設定できるネットワーク・ワイドな名前の二階層の名前空間を持ち、ユーザから見た使い易さと、大規模なネットワークにおける管理の容易さを同時に達成した。また、データ共有のセマンティクスとしてセッション・セマンティクスを採用し、ローカル・キャッシュによる速度の向上を目指した。これによって、複数のバージョンを管理する必要が生じるが、仮想記憶の機構を利用して自然に実現できるので、そのためのオーバーヘッドは少ない。

The File System in the distibuted OS ToM

Hideki Tsuiki　Keio University
Takashi Hattori　Kyoto University
Masaki Harima　Nippon Data General

The brief description of the ToM file system is given. First, it has two-level name space; one is local in each machine, and the other is network-wide which can be configurable for each user. The two-level name space is suitable for both access by users and maintainance in a large-scale network. Second, the ToM file system adopts session semantics for data sharing, in order to increase efficiency by using local cache. Although this causes request to maintain multiple versions, it is smoothly managed by a mechanism of the virtual memory in ToM so that its overhead is small.

1　はじめに

ワークステーションとネットワークが普及するにつれて、従来のTSSと端末を前提とするUNIXなどのオペレーティング・システムの限界が明らかになってきた。この新しい環境に対応するため、MachやSpriteなど、ネットワーク上の資源を有効に利用する分散OSの開発が盛んになりつつある。そこで、高速のネットワークによって接続された計算機群を前提とし、新しいプログラミング・モデルに基づく分散OS "ToM" の研究、開発を行なってきた[1]。ToMは

- 分散環境に適したプログラミング・モデルを提供する。
- 完全にネットワーク透過な分散環境を提供する。
- 複数機種が存在するネットワークをサポートする。
- ネットワーク構成の変化に容易に対応できる。
- ネットワーク上でのセキュリティが高く、かつユーザが使いやすい環境を構築する。
- ユーザごとに最適な環境を定義できるようにする。
- UNIX 4.3BSD のシステム・コールをエミュレートする機能を提供する。

などを目標に設計を行なった。

ここでは、ToM のプログラミング・モデルについて簡単に述べた後、ファイル・システムの概要を解説する。さらにToMファイル・システムの特徴の一つである、file view 機能と、データ共有セマンティクスについて詳しく解説する。

2　ToMのプログラミング・モデル

2.1　スレッドとモジュール

UNIXなどのシングルプロセッサのOSでは、プログラムと、それを実行する処理の流れを一体化してプロセスとしていた。これは、一つのメモリ空間と一つのプロセッサしかない基本的な計算機をモデル化したものである。このため、複数のプロセッサが同一のプログラムを実行する共有メモリ型マルチプロセッサはうまく取り扱うことができなかった。これに対してMachでは、プロセスをタスクとスレッドに分解し、複数のスレッドが一つのタスク内で走ることを許すことによってマルチプロセッサを取り扱うことに成功した。ToMはこれをさらに推し進めて、プログラムを抽象化したモジュールと、処理の流れを抽象化したスレッドを完全に独立させて、複数のメモリ空間と複数のプロセッサが存在する分散環境をモデル化する。

スレッドは後述するRPCを用いることにより、現在実行しているプログラムが存在するモジュールを離れ、新しいモジュールに移ることができる。これは、スレッドの観点からすると、複数のモジュールをRPCで動き回ることによって処理を行っているように見える。逆にモジュールの観点からすると、複数のスレッドがRPCで飛び込んできて処理をしているように見える。すなわち、ToMのプログラミング・モデルでは、複数のモジュールと複数のスレッドが連係をとりながら処理を行なうことを可能としている。

2.2　RPC

ToMでは基本的な通信手段としてRPC (リモート・プロシージャ・コール) を採用した。RPCではリモートの処理が終るまで呼び出したプロセスは停止し、リモートの処理結果を受け取ってから再開する。 RPCをプリミティブとした理由は、以下の三点である。

- プロセス間通信の多くが同期型であること。
- RPC向きの通信プロトコルを用いることにより、非同期型の通信でRPCを実現することに比べ高速化やエラー処理などが容易になる。
- RPCは手続き呼び出しの形のため使いやすい。

さらに、ToMにおけるRPCは、スレッドとモジュールが独立しているため、他のスレッドに手続きの実行を依頼するのではなく、そのスレッドが実行しているモジュールが切り替わってそのまま実行するという形をとる。

[1]ToMの研究・開発は、京都大学、慶應義塾大学、京都高度技術研究所が企業(アステック、エスアールエー、オムロン、キヤノン、三洋、ソニー、東芝、日本鋼管、日本サン・マイクロシステムズ、日本・データゼネラル、日本ビクター、ビクター・データ・システムズ、富士ゼロックス、横河・ヒューレット・パッカード)と共同してすすめている。

この方式は、別のプロセスを起動するのに比べて、プロセスの生成や再スケジューリングが無い分、負荷が軽いと考えられる。また、スレッドが停止して待つ必要がないので、スケジュールで割り当てられたCPU時間を有効に使うことが期待できる。さらに、同一計算機上のモジュールにRPCを行う場合には、仮想空間もモジュールの部分が切り替わるだけで、スレッドの部分は変える必要がない。また、別のプロセスを起動した場合には、それぞれのプロセスの処理の流れと、RPC で呼び合う情報の流れの三つの流れがあったのに対して、この方式では、一つの流れが存在するだけであり、より自然に分散処理を表現することができると考えられる。

3 ToMファイル・システムの概要

ToM ではカーネルをなるべく小さくする方針をとっており、ファイル・システムもユーザ・レベルで動くサーバの一つに過ぎない。このことは、ファイル・システムは固定されたものではなく、ユーザによって交換できるということを意味している。むしろ、さまざまな環境に対応するために、ファイル・サーバを複数のモジュールで構成し、その一部を交換することを積極的に支援している。したがって、ここで述べる機能も、ToMにおける標準的なファイル・サーバの場合であり、ユーザが新たにファイル・サーバを作って変更することが可能である。

3.1 特徴

3.1.1 二階層の名前空間

ディスクは branch という単位で管理される。ディスク上に記録される名前(path name)は、branch の中で木構造をなしている。Branch 相互間には何の関係もない。それに対して、ユーザがアクセスするのに使う名前(view name) は、ネットワーク上に存在する branch すべてにまたがるような単一の木構造を与える。ここで重要なのは、view name はファイルの物理的位置から完全に独立していることと、ユーザごとに異なる view name を使えることである。View name については後の 4 file view のセクションで詳しく述べる。

3.1.2 マッピングによるアクセス

基本的なアクセス方法として、ファイルをアドレス空間にマッピングする機能が提供される。マップされた後は、その番地に対する読み書きがファイルに対する読み書きとなる。ファイルが実際に存在するのが別のマシンであっても、完全に透過的にアクセスできる。

3.1.3 セッション・セマンティクス

データ共有セマンティクスとして、セッション・セマンティクスを採用した。この場合の問題点として、ファイルの更新が行なわれた時に、古いバージョンをアクセスしているプロセスがある限り、そのバージョンを保存しておかなくてはならない。ToM では、このオーバーヘッドを減らすために、完全な情報を持つバージョンは常に一個だけで、その他のバージョンは差分だけを保持する。この機構はメモリ管理の機能を利用して効率よく実現できる。これについては、5データ共有セマンティクスのセクションで詳しく述べる。

3.1.4 アクセス・コントロール・リスト

ファイルに対するアクセスは、アクセス・コントロール・リストによって制限される。アクセス・コントロール・リストの要素は名義と呼ばれる。名義は、個々のユーザだけでなくユーザのグループも表すことができる。

3.1.5 複製

可用性と速度の向上のため、ファイルの複製を持つことができる。複製は、branch 単位で生成され、読みだし時には複製の中から一つを自動的に選んでアクセスする。それに対して、書き込みは常にオリジナルに対して行なわれ、それと同時にその時点でアクセス可能なすべての複製が更新される。

3.1.6 属性変数

ファイルには、その属性を示すために、キーと値の組が付加されている。キーと値はどちらも文字列で、サイズや生成日時などファイル・サーバが管理する特定のキー以外は、ユーザが自由に設定できる。

3.2 構造

ファイル・サーバはユーザ・レベルのプログラムであり、ユーザによって交換が可能である。標準的なファイル・サーバの構造は、次のようなモジュールから成り立っており、ユーザのスレッドがRPCによって各モジュールを実行する。

- Upper File Manager

  View name から path name への変換を行ない、そのファイルが存在するbranch を特定して、そこを管理する Lower File Manager を呼び出す。

- Lower File Manager

  Branch の管理を行なう。また、path name によるアクセス要求を受け付けて、ファイルごとに File Mapper を作り出す。

- File Mapper

  アドレス空間にマップされたファイルに対して、主記憶と二次記憶装置の間の転送を制御する。

- Storage Manager

  ディスクを管理し、低レベルのファイル・アクセスを制御する。

4 File View

4.1 分散ファイル・システムにおける名前付け

分散ファイル・システムにおける名前付けに関して、考慮すべき点として次の二つが挙げられる。

- ファイル位置からの独立

  ファイルの存在する物理的位置とは無関係に名前を付けることができる。

- ユーザ位置からの独立

  ユーザがあるマシンから別のマシンへ移動しても、ファイル名は変わらない。

NFS[6], RFS[5] などは、ファイル位置からは独立しているが、マシンごとにマウント情報を管理しているので、ユーザ位置からは独立していない。逆に Apollo Domain[4] などはユーザ位置からは独立しているが、ファイルの存在するマシン名がファイル名に含まれるので、ファイル位置からは独立していない。ファイル位置からもユーザ位置からも独立しているものとして、Sprite[2] や Andrew[7] がある。

Sprite も Andrew も、ネットワーク上に存在するすべてのファイルを含むような、単一の木構造を提供している。しかし、この方式では、ネットワークが大規模になってくると次のような問題点が出てくると考えられる。

- 木構造に含まれるファイルの数が極めて多くなるので、実際に各ユーザが使用するファイルは全体に比べればほんの一部分だが、衝突を避けるためにユーザ名を含めたり、長いパス名を使用したりしなくてはならない。

- 各マシン上の部分木が、互いにどのように接続されるかを管理するのが複雑になる。特に、ネットワークの構成が頻繁に変更される場合は、そのたびに管理者がファイル・システムの構成をそれに対応させなくてはならない。

4.2 Path name と view name

ToM ではファイルの名前付けには二つの階層がある。まず、物理ディスクはいくつかの branch に分割される。それぞれの branch の中で、UNIX と同じように path name が定義される。したがって、branch name と path name の組によって、ファイルを一意的に識別することができる。このレベルでは、ファイル位置からの独立は達成されない。

ユーザがアクセスするのに使うのは、view name である。View name は、あらかじめ与えられた変換テーブルによって、path name に変換されてから実際のアクセスが行なわれる。変換テーブルは複数のユーザで共有することもできるし、一人のユーザが複数のテーブルを定義することもできる。ユーザは path name を意識することなく、view name で構成される単一の木だけを扱う。この木を、そのユーザの file view と呼ぶ。

例えば、jun, sys, public という branch がある時に、次のような変換テーブルを与えたとする。

/bin	jun:/bin	<sys:/local/bin
/src	jun:/src	>public:/src

左端には view name の prefix を書く。その右には、path name の prefix の列を書く。View name が与えられると、最長一致する prefix を選び、prefix の部分を右に書いてある prefix で順に置き換えて、そのファイルが存在するかを調べる。最初に発見されたファイルが、アクセスの行なわれるファイルになる。例えば、/bin/echo という view name に対しては、jun というbranch の /bin/echo をまず探し、無ければ sys という branch の/local/bin/echo を探す。したがって、このテーブルによる file view は、最上位に /bin と /src があり、その下にそれぞれ jun:/bin と sys:/local/bin をマージした部分木、jun:/src と pulic:/src をマージした部分木がある。

Path name の prefix の列の最初の一個は primary directory と呼び、必ずディレクトリでなくてはならない。二個目から後は、ディレクトリでもファイルでも良いが、前に < か > を付けて、更新の方法を指定しなければならない。上のテーブルの一行目では、sys:/local/bin の前に < が付いている。jun:/bin/echo が存在せず、sys:/local/bin/echo が存在する時、/bin/echo を書き換えようとすると、sys:/local/bin/echo は変更されず、primary directory の jun:/bin にコピーが作られて、それが変更される。これに対して、二行目の public:/src のように > が付いていると、コピーは行なわれず、public:/src に存在するファイルが変更される。この機能は、前者が copy on write、後者が symbolic link に相当する。これによって、バージョンごとにディレクトリの差分を作る場合なども簡単に管理できる。

Path name と view name による二階層の名前は、次のような利点があると思われる。

1. File view を適当に構成することにより、不要な部分を見せないようにできるので、全体の規模が大きくなっても簡潔な名前でアクセスできる。

2. ユーザごと、あるいは作業の内容ごとに file view を変更できるので、link の必要性が少なくなり、複雑な symbolic link で混乱することが防げる。

3. 空のディレクトリを primary directory とし、ソース・コードの入っているディレクトリを copy on write で同じ view name に重ねておくと、変更したファイルは primary directory にコピーされるので、バージョン管理が容易にできる。

4. アプリケーションの初期設定ファイルも、標準のファイルから copy on write で各ユーザ用にカスタマイズしたファイルを作ることができる。この時、view name は変わらないので、アプリケーションがサーチ・パスを持つ必要がない。

5. 管理者は branch と path name のレベルで管理することができる。Branch はそれぞれ独立しているので、ネットワークの規模が大きくなっても管理が複雑になることはない。ネットワークに新しいマシンが追加された時は、branch の管理テーブルの追加だけで良いし、バックアップなども branch ごとに行なうことができる。

このうち、2, 3, 4は分散環境でない場合でも有用であり、TOPS-20 や TFS[3] などでも同様のことが行なわれている。しかし、5は分散ファイル・システムを管理する時の利点であり、ユーザが使用する名前とは別の階層の名前を持つということが本質的に重要である。

5 データ共有セマンティクス

5.1 ToM におけるセマンティクス

分散ファイルシステムでは、ファイルを複数のプロセスが同時にアクセスしている時に、ファイルの更新がお互いにどのように影響するかが問題になる。[1]によれば、次の三種類の方式に分類できる。

- UNIXセマンティクス

  メモリの共有と同様に、どれかのプロセスが内容を書き換えた時には、そのファイルを読み書きしている他のプロセスに、同時に反映される。これは、単一計算機上でファイルシステムを実現する場合には、もっとも単純で、素直な方法である。しかし、他のプロセスが書いている途中のファイルの状態が見えてしまうため、アプリケーション・プログラムがロックなどの操作を行なわなければならない。また、複数のマシンで同時に書き込みが行なわれる場合、変更を互いに反映させるための通信量が多くなる。

- トランザクション・セマンティクス

ファイルの内容を更新している途中のファイルの状態は、他のプロセスに見せては問題が起こる。そこで、書き込みを行なっているプロセスがコミット(書き込みの終了)を行なうまでは、他のプロセスの見ているファイルの内容は更新を行なわないという方法である。これは、データ・ベースの更新と同様の方法である。継続してファイルを読んでいるプロセスにとっては、他のプロセスのコミットによりファイルの内容が突然変わることになるので、何らかの同期操作が必要になる。

- セッション・セマンティクス

  ファイルのアクセスを開始した時点で、そのプロセスから見えるファイルの内容は固定され、他のプロセスからの書き込みにより影響されることはない。ファイルの更新は、アクセスを終了した時点で他のプロセスに公開され、それ以後にアクセスを開始したプロセスは、更新後のファイルを読むことができる。この方法では、複数のプロセスが独立にファイルにアクセスする時には、まったく同期する必要がない。逆に同期して動くプロセス間では、アクセスをいったん終了してから再開することにより、他のプロセスによる更新を反映させることができる。また、分散環境においては、ファイルの更新を他のマシン上のキャッシュに反映させる必要がないため、通信量を大幅に減らすことができる。

ToM では、通信量を重視して、セッション・セマンティクスを採用した。これにより、各マシンにおけるファイル・キャッシュは独立に管理することができ、大幅な速度の向上が見込める。その反面、ファイルの更新が行なわれた時に、古いバージョンをアクセスしているプロセスがある限り、そのバージョンを保存しておかなくてはならない。この時にファイル全体をコピーするのは、速度とディスク領域の両方で損失であるから、完全な情報を持つバージョンは常に一個だけで、その他のバージョンは差分だけを保持するようにする。ところが、ファイル・アクセスはすべてメモリ・マッピングによって行なわれるので、バージョン管理は主記憶上とディスク上のそれぞれで行なわなければならず、しかも仮想記憶の機構と整合している必要がある。そこで、ToM の仮想記憶には、バージョン管理を最小限のオーバーヘッドで実現するための機能が含まれている。以下では、その機能と、ファイル更新時にどのようにしてバージョンが生成、消滅するかを説明する。

5.2 仮想ページ

ファイルをアドレス空間にマップすると、カーネル内に仮想ページと呼ばれる構造が作られる。仮想ページはファイルに限らず、仮想記憶空間上に存在するすべてのデータに対して作られ、アドレス空間に仮想ページを対応させることによって、プロセスがそのデータにアクセスできるようになる。また、仮想ページにはそれぞれページャが対応しており、ページングが必要になった時は、そのページャが主記憶とディスクの間の転送を行なう。ファイルの場合のページャは file mapper である。

仮想ページには、メモリ共有や copy on write を実現するためにシャドウという概念がある。ある仮想ページに対してシャドウ仮想ページを作ることができ、シャドウ仮想ページはオリジナルとの差分だけを保持している。また、ファイルのバージョン更新を容易に実現するため、シャドウ関係を逆転させることができる。シャドウ関係を逆転させると、オリジナルからシャドウへ共通部分が受け渡され、元のオリジナル仮想ページの方が差分だけを保持するようになる。

5.3 ファイル更新時の動作

ファイルを read mode で open すると、仮想ページAが作られる。さらに別のプロセスが read mode で open しても同じ仮想ページが使われる。それに対して、read/write mode で open した時は、Aに対してシャドウ仮想ページA' が作られる。A'は最初は空であり、アクセスやページングはAの保持しているページに対して行なわれる。A'に対して書き込みが行なわれると、そのページはA'にコピーされてから書き込みが行なわれる(図1)。A'の行なうページングは、A'が保持しているページだけであり、ディスク上に余分な領域は必要としない。

A'をアクセスしているプロセスが close を行なうと、A'の内容が新しいファイルの内容となる。そのために、AとA'のシャドウ関係を逆転させ、AがA'のシャドウ仮想ページになるようにする。この時、変更されなかったためにA'にコピーされていないページは、AからA'へ移動する(図2)。この操作はカーネル内のテーブルの変更だけで、実際のメモリ間の転送は起こらない。ま

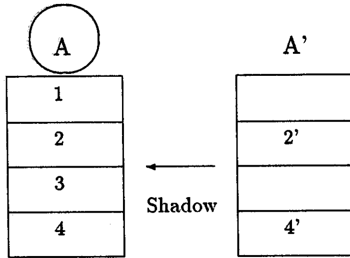

図 1: A'がページ2,4に書き込みを行なった状態(ファイルを read open するとAの内容が見える)

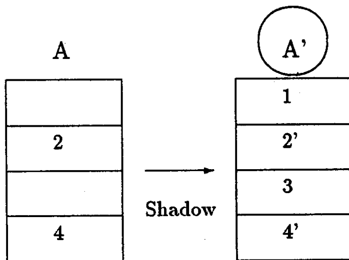

図 2: A'がcloseされた後の状態(ファイルを read open するとA'の内容が見える)

た、それに対応して、ディスク上でも古いバージョンのディスク・ブロックと、A' からページアウトされたディスク・ブロックで新しいファイルが構成され、古いバージョンのディスク・ブロックで更新されてしまった部分(新しいバージョンの一部として使われない部分)は、Aのページング領域として使われるようになる。これ以後、read mode で open したプロセスはA'を使い、read/write mode で open したプロセスはさらにA"を作って同様に進んでいく。

使われなくなった仮想ページを検出するために、リファレンス・カウンタを使用している。カウンタは、その仮想ページをアクセスしているプロセスの個数と、その仮想ページをオリジナルとするシャドウ仮想ページの個数の合計を記録している。したがって、上の例で、A'がcloseされるまではAをアクセスしているプロセスがなくてもAは残っている。そして、A'がcloseされると、シャドウ関係が逆転するのでカウンタの値が0になり、AとAが使っているページング領域は解放される。この時、Aをアクセスしているプロセスが存在すれば、そのプロセスがcloseした時点でカウンタが0になるので、それまでAは残ることになる。

一般には read/write mode の open は並行していくつでもできるので、仮想ページのシャドウ関係は木構造になる。この木の性質として、read/write mode に対応する仮想ページは葉にしか現れず、それ以外の節点は必ず read mode であり、シャドウ関係のオリジナルをたどっていくと根に到達する。ある時点でのファイルの内容は、根の仮想ページによって表される。また、read/write mode のアクセスがclose されると、それに対応する葉から根に至る道筋でシャドウ関係の逆転が起こり、close された仮想ページが新しい根となる。

このように、ファイルの更新によって生じた複数のバージョンは、それぞれに対応する仮想ページ間のシャドウ関係によって自然に実現できる。この方式の利点として、

- 主記憶上でもディスク上でも共有できるページは必ず共有され、無駄な領域やコピーは必要としない。
- ファイルをそのままページング領域として使えるため、新しいページング領域を確保する必要がなく、仮想記憶との整合性が良い。
- 複数のバージョンが存在している時点でシステムがダウンした時は、最後に更新を終了したバージョンがファイルとして残り、それ以外のバージョンはゴミとして回収される。

などが挙げられる。

6 おわりに

最初のバージョンではUNIX ファイル・システムを利用して、その上に ToM ファイル・システムを載せる。現在は Upper File Manager と Lower File Manager のコーディング中であり、1990年秋完成予定である。その後、UNIXの部分を置き換えて、完全なファイル・システムにする予定である。

謝辞

京都大学数理解析研究所の中島玲二教授を始めとする、分散OSプロジェクトのために多大な努力を

払っていただいた方々に深く感謝の意を表する。

参考文献

[1] Levy, E., Silberschatz, A., *Distributed File Systems: Concepts and Examples*, TR-89-04, Department of Computer Science, the University of Texas At Austin, 1989.

[2] Ousterhaut, J.K., et al., *The Sprite Network Operating System*, IEEE COMPUTER, vol.21 no.2, pp.8-22, 1988.

[3] Hendricks, D., *A File system For Software Development*, Proc. Summer Usenix Conf. pp.333-340, 1990.

[4] Levine, P., *The Apollo Domain Distributed File System*, Theory and Practice of Distributed Operating Systems, NATO ASI Series, Springer-Verlag, 1987.

[5] Fifkin,A.P., et al., *RFS Architectural Overview*, Proc. Summer Usenix Conf., pp.248-259, 1986.

[6] Sandberg, R. et al., *Design and Implementation of the Sun Network File System*, Proc. Summer Usenix Conf., pp.119-130, 1985.

[7] Satyanarayanan, M., *Scalable, Secure, and Highly Available Distributed File Access*, IEEE COMPUTER, vol.23, no.5, pp.9-21, 1990.