オペレーティング・システム 45 − 2
（1989. 11. 10）

分散 OS ToM

— 新しいプログラミングモデルとセキュリティ機構について —

新井潤　桜川貴司　立木秀樹
京都大学数理解析研究所

萩野達也
京都大学大型計算機センター

服部隆志
慶應義塾大学

森島晃年
京都高度技術研究所

高速ネットワークで接続された計算機群を利用するための分散OS ToM (Threads on Modules) の概要を説明する。ToM は、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独立したものとして扱い、通信として RPC (Remote Procedure Call) を用いることにより、分散環境にふさわしいプログラミング・モデルを提供している。ToM の RPC は従来のものとは異なり、スレッドとモジュールの組合せを変更することにより行なわれる。また、ケーパビリティとアクセス・コントロール・リストを組み合わせたセキュリティ機構についても述べる。

A Distributed Operating System ToM

— its programming model and security —

Jun ARAI　Takashi SAKURAGAWA　Hideki TSUIKI
RIMS, Kyoto University
Oiwake, Kitashirakawa, Sakyo, Kyoto 606, Japan

Tatsuya HAGINO
KUDPC, Kyoto University
Hon-machi, Yoshida, Sakyo, Kyoto 606, Japan

Takashi HATTORI
Keio University
Mita, Tokyo, Japan

Akitoshi MORISHIMA
ASTEM/RI Kyoto
Minami-machi, Chudoji, Shimogyo, Kyoto 600, Japan

A new distributed operating system ToM for computers connected by a high speed network is presented. ToM provides a new programming model which is suited for distributed environment. Programs are modeled as modules and control flows are modeled as threads, and they are combined arbitrarily. A module may have more than one threads running in parallel. A thread may execute codes in more than one modules. A thread can move from its currently-running module to another module by RPC (Remote Procedure Call). ToM's RPC is performed by just changing combination of modules and threads. It does not need conventional client and server processes. We also describe ToM's security control. It combines capability method and access control list method.

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[1]やSprite[8]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。

ToM[1]は、高速のネットワークによって接続された計算機群を統一的に操作するために、新しい計算モデル基づいて設計された分散OSである。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。

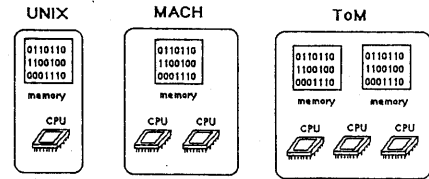
などのことを目標に設計が進められている。

Machなどの最近の分散OSに見られるように、CPUが特権レベルで動くUNIXのカーネルに相当する部分(ToMではcoreと呼んでいる。)は最小限に抑え、機能の大部分をユーザ・レベルのプログラムで実現する方針をとっている。よって、ファイル・システムなど、OSの基本的な機能もユーザ・レベルのプログラムとして実現される。

ユーザ・レベルで分散プログラムを実現するためには、coreが分散プログラムに適したプログラミング・モデルを採用している必要がある。今までのOSが提供するプログラミング・モデルでは、分散環境上の資源を有効に利用するアプリケーションの作成が難しかった。それは、単一のコンピュータ上でアプリケーション・プログラムが動作することを前提としているからである。ToMでは、ネットワーク透過な環境の下で、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分離し独立したものとして扱い、通信としてRPC (Remote Procedure Call)を用いることにより、分散環境にふさわしい計算モデルを提供している。ToMのRPCは、従来のものとは異なり、スレッドとモジュールの組合せを変更することにより行なわれる。

分散プログラムは、それぞれのモジュールがRPCで他のモジュールを呼ぶことにより実現される。その際に、どのモジュールに対してでもRPCをすることができるのではセキュリティ上問題があるので、RPCをする権利を制限する必要がある。この権利は、動的に変更可能でなくてはならない。また、モジュールやスレッドに対しては、それらを止めたり破壊したりといった制御の必要が生じる。この制御の権利は、RPCの権利と違って静的に与えられるべきである。ToMでは、前者の権利はケイパビリティで、後者の権利はハウスとユーザという概念で制御している。

本稿では、ToMのcoreが提供している機能のうち、プログラミング・モデルおよび、セキュリティに関して概説する。



UNIXは一つのプログラムと一つのプロセッサ、Machは一つのプログラムと複数のプロセッサ、ToMは複数のプログラムと複数のプロセッサをモデル化している。

図1: プログラミング・モデル

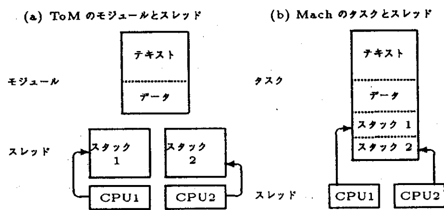
2 プログラミングモデル

UNIX[11]など、単一計算機のOSでは、プログラムとプロセッサ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[1]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では共有メモリ型マルチプロセッサだけでなく、分散環境をもうまくモデル化することを目的としている。そのため、ToMではプログラムを抽象化したモジュールと、プロセッサを抽象化したスレッドを完全に独立させた。Machの場合、スレッドはタスクに属するものでタスクの中に閉じ込められていたが、ToMの場合にはスレッドはある一つのモジュールに属しているのではなく、複数のモジュールを渡り歩くことができる。スレッドは、後述するRPCを用いることによって、現在プログラムを実行してい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見れば、複数のモジュールをRPCで行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見れば、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

[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と共同してすすめている。



ToM では、他のスレッドのスタックはアクセスできない。Mach では、他のスレッドのスタックも同じタスク上にあるので、破壊してしまう危険性がある。

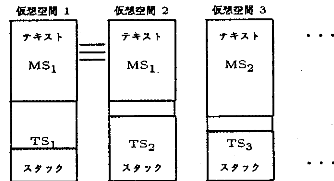
図 2: スレッド空間

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一体となって処理を行う。複数のモジュールによってネットワークに分散したプログラムをモデル化することができ、複数のスレッドによって密結合マルチプロセッサおよびネットワークに分散したプロセッサをモデル化することができる。UNIX のプログラミング・モデルは ToM における一つのモジュールと一つのスレッドの組みに対応し、Mach のプログラミング・モデルは ToM における一つのモジュールと複数スレッドの組みに対応する (図 1)。

ToM では、モジュールとスレッドのどちらにもメモリが割り当てられる。それぞれ**モジュール空間** (MS)、**スレッド空間** (TS) と呼ばれる。モジュール空間には、通常、命令コードとデータが置かれ、スレッド空間にはスタックが置かれる。Mach のスレッドは空間を持たず、スタックはタスクに置かれていて他のスレッドからアクセスされる可能性があった。スタックはそのスレッド固有のものであり共有できないのであるから、スタックはスレッドの一部であると考えるのが自然であろう (図 2)。スレッドが現在プログラムを実行しているモジュールをそのスレッドのカレント・モジュールという。スレッドは RPC を行うことによってカレント・モジュールを切り替えることができる。また、複数のスレッドが同一のモジュールをカレント・モジュールにしている場合もある。実際の計算機上で実行する時は、スレッドの TS とそのスレッドのカレント・モジュールの MS を CPU の論理アドレス空間にマップする。

モジュールには、**エントリ**と呼ばれる手続きの入口が複数存在し、RPC はこのエントリを指定して行なわれる。エントリは、モジュール空間のアドレスと RPC を行なう際の引数の型と返り値の型の情報からなっていて、モジュールを作成した時に宣言し登録しておく。一つのモジュールが複数のエントリを持つことができ、それぞれ異なった処理をする手続きの入口にすることができる。

一般に、一つのモジュール上を複数のスレッドが走ることができるので、それらの間で共通資源の管理等の排他制御を行なう必要がある。そのために、排他制御型の



MS_1 と TS_1、MS_1 と TS_2、MS_2 と TS_3 は、それぞれ同じ仮想空間にマップされている。こういったコンテキストが複数あって並列に実行している。また、スレッド 1 とスレッド 2 は、同じモジュール 1 上を並列に走っている。

図 3: モジュール空間とスレッド空間

モジュールが存在する。あるスレッドがすでにこの型のモジュール上を走っているのに、他のスレッドが RPC を行なって同じモジュールに飛び込もうとした時には、そのスレッドは一時的に停止させられ、すでに走っていたスレッドがモジュールを抜けるまで待たされる。

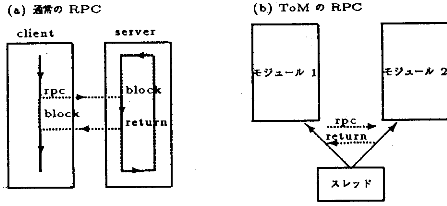
3 ToM の RPC

ToM ではモジュール間の通信手段として **RPC** (リモート・プロシージャ・コール)[4] を採用した。RPC はリモートの手続きをローカルなプロシージャ・コールと同じようにして呼び出す機能である。UNIX のソケットや Mach のポートのようにメッセージの列を転送して非同期的に処理を行うのではなく、RPC は同期的である。RPC ではリモートの処理が終わ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三つある。一つはプロセス間通信の多くが同期型であることである。もう一つは、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 を実現するのに比べ高速化やエラー処理などが容易になることである [5]。最後に、RPC は手続き呼び出しの形のため使い易いのは言うまでもない。

さらに、ToM における RPC は、通常のものとは異なる。あるモジュールから他のモジュールの手続きをサブルーチン・コール的に呼び出すといった意味では普通の RPC と変わりはないが、スレッドとモジュールは独立しているため、他のスレッドに手続きの実行を依頼するのではなく、そのスレッドのカレント・モジュールが切り替わってそのまま実行する (図 4)。

普通の RPC では、クライエント・プロセス A とサーバ・プロセス B があり、クライエント A が RPC の要求を出すとサーバ B が要求を受け取り処理をして値を返す。これに対して、ToM ではモジュール A 上を走っているスレッドがモジュール B に RPC をする時、単にカレント・モジュールを A から B に切り替え適当なエントリに飛び込んで自分自身が B 上を走って処理を行う。処理結果が得られるとカレント・モジュールを元の A に戻して処理を続行する。この方式は、別のプロセスを起動する



UNIX では 2 つのプロセスが同期を取り合って RPC を行なったが、ToM ではスレッドが単にカレント・モジュールを切替えることにより RPC を行なう。

図 4: ToM の RPC

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 MS が切り替わるだけで TS 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。

4 ハウス

モジュールとスレッドは、ハウスと呼ばれる単位でグループを作っている (図 5)。ハウスは、一つの仕事をしているモジュールとスレッドの集まりであるという意味で UNIX のプロセスに似ている。2で説明したプログラミング・モデルそのままでは、多数のモジュールとスレッドが計算機上に混在するため管理などが大変である。ハウスはあるまとまりのある仕事を行っているモジュールとスレッドを一つにしたもので、モジュールやスレッドの制御 (作成や消去など) の権利を表わす単位にもなっている。これは、 UNIX のプロセスに相当する。

ハウスには、その所有者であるユーザの集合が対応づけられている。通常のハウスではこの集合はただ一人のユーザからなり、そのユーザのスレッドだけが制御できる。協同作業を行う場合には複数のユーザからなる集合がハウスに対応づけられるが、その場合にはこの集合に属する任意のユーザのスレッドがこのハウスを制御することができる。

スレッドの持つ制御権は、そのスレッドの属するハウス (後で述べるようにスレッドはハウスを指定して作成される) と、そのスレッドのカレント・モジュールの属するハウス (モジュールもハウスを指定して作成される) によって決められる。この二つのハウスのユーザ集合の和集合をこのスレッドの**実効ユーザ集合**と呼ぶ。例えば、図 5で、スレッド 4 がモジュール 3 を実行している場合の実効ユーザ集合は、{ 1,2,4 }となる。カレント・モジュールは RPC によって変化するので、実効ユーザ集合も RPC によって変化する。

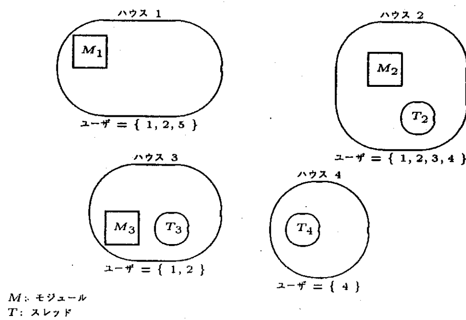
実効ユーザ集合は大きくなるほどその制御権が弱くなる。例えば、図 5のモジュール 3 を実行しているスレッド 4 はユーザ 1、ユーザ 2 およびユーザ 4 が共に所有しているハウス 2 に対してのみ制御ができる。ハウス 1 やハウス 3 に対しては制御できない。スレッドの実効ユーザ集合をAとした場合、ユーザ集合BがAより大きいか等しい ($A \subseteq B$) ハウスに対してのみこのスレッドは制御権を持つ。したがって、実効ユーザ集合が空であるスレッドは全てのハウスに対する制御権を持つことになり、 UNIX で言うスーパーユーザの権限を持つプロセスに対応する。逆に、ユーザ集合が空のハウスを制御できるのはこのようなスーパーユーザ権限を持つスレッドだけに限られる。

スレッドにはそのスレッドが作成された時に指定された**リアル・ハウス**とは別に、システム・コールによって設定された**実効ハウス**があり、スレッドの実効ユーザ集合は実効ハウスによって計算される。これは、 UNIX の SUID (Set User ID) 機能と同じようにハウスに RPC して来たスレッドの権限を一時的に変更するために使われる。

ハウスが作られた時、そのハウスのユーザ集合は、作ったスレッドの実効ユーザ集合になる。ハウスを壊したり、ハウス内のスレッド、モジュールを作成、破壊したり、スレッドを一時停止や再開する時には、そのハウスに対する制御権を持っていなくてはならない。ハウスのユーザ集合は、システムコールで増やしたり減らしたりすることができる。ユーザ集合を増やすには、ハウスの制御権を持っていればよいが、ユーザ集合を減らすには、減らした後できたハウスに対する制御権を持っていなくてはならない。例えば、{ 1,2,3,4 }という 4 人のユーザが所有するハウスから 4 を減らす場合、実効ユーザ集合が{ 1,2 }であれば ({ 1,2,3,4 }から 4 を取った){ 1,2,3 }というユーザ集合に対しても制御権を持つので可能であるが、実効ユーザ集合が{ 3,4 }であると{ 1,2,3 }に対して制御権を持たないので失敗する。つまり自分で自分を減らす (例えば、{ 4 }から4を減らしてスーパーユーザにしようとする) ことはできない

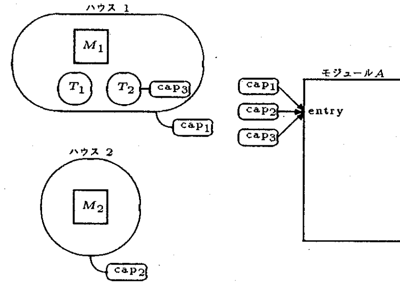
スレッドはハウスに対する制御権を持っている時に、モジュールを作成したり、スレッドを作成したりすることができる。スレッドを作成する時にはモジュールとエントリを指定し、作成されたスレッドはそのエントリから走り出す。モジュールやスレッドの破壊もそれらが属するハウスの制御権を持っているスレッドが行なえる。この他、モジュールのエントリの登録やスレッドの一時停止や再開などの制御も可能である。

このように、ハウスのユーザの集合とスレッドの実効ユーザ集合によってモジュールやスレッドは制御される。ただし、 RPC を行う権利に関しては、 5で説明するケイパビリティによって制御される。



M_3とT_3の組合せならハウス2に対する制御権がある。M_1とT_3の組合せでは、実効ユーザ集合に5が入るので制御権がない。また、M_3とT_3は、ハウス2のユーザから4を減らすことができるが、M_3とT_4にはできない。

図5: ハウスとユーザ



モジュールAの entry には、3つのケイパビリティがある。今、T_1とT_2はM_1上を走っているとして、どちらも entry へ RPC できる。ところが、cap_1を消す (破壊する) と、T_1は、entry へ RPC できなくなるが、T_2は自分でcap_3を持っているので RPC できる。また、この場合でもT_1がM_2上を走っているのであれば、cap_2によって entry へ RPC できる。

図6: ケイパビリティ

5 ケイパビリティ

RPC を行う権利をハウスに関する制御権で行うと、少し不都合が起きる。例えば、ファイル・システムを実現しているハウスの所有者を空集合にすると一般のユーザは RPC をすることができなくなり、ファイルが取り扱えなくなる。逆に、所有者を全員にすると誰もがファイル・システムのハウスのモジュールを破壊できたりして危険である。このため、ToM では RPC を行う権利の管理をハウスの制御権で行うのではなく、ケイパビリティという別の機構で行う。

ケイパビリティは、あるモジュールのあるエントリに RPC を行うことのできる権利を表わしている。ケイパビリティはシステムコールでモジュールとエントリを指定して作られる。一つのエントリに対して複数のケイパビリティを作成してもよく、同一エントリに対する RPC であってもケイパビリティによって動作を変えたり、選択的にケイパビリティを奪うことができるようになっている。例えば、ファイル・システムではケイパビリティをファイル・ディスクリプタに使うことができる。個々のオープンしたファイルに別々のケイパビリティを与えておき、ファイルの読み書きに関しては同じエントリに RPC して来るがケイパビリティの違いによって読み書きするファイルを決定することができる (一種のデータ隠蔽を実現していることになる)。

スレッドおよびハウスに対応して、それらが持っているケイパビリティの集合が core 内に保持されている。スレッドは自分自身がケイパビリティを持っているか、カレント・モジュールの属するハウスがケイパビリティを持っている時に、そのケイパビリティの対応するエントリに RPC をすることができる (図6)。

スレッドの持つケイパビリティのうち、あるモジュールを実行中に有効なものは、そのモジュールに対する RPC の引数として渡されたものであるか、そのモジュールからの RPC またはシステムコールの返り値として受け取ったものかのどちらかである (図7参照)。

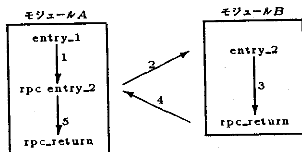
ハウスが持つケイパビリティは、システムコールで明示的に与えられたものである。通常は、RPC の返り値としてスレッドが得たケイパビリティを後で利用するためにカレント・モジュールのハウスに自分で与えておく場合が多い。ファイル・サーバなどのみんなで共通で使うケイパビリティなどは、ネーム・サーバーに依頼して与えてもらうようにすることもできる。

6 まとめ

分散 OS ToM の core が提供するプログラミング・モデルおよびセキュリティの機構について述べた。これらの機構は、それぞれの計算機上で実現されるだけでなく、ネットワーク上で透過に実現される必要がある。ToM では、まず、それぞれの計算機上でローカルな core を実現し、その上で、ネットワーク 透過な core をユーザ・レベルのプログラムとして実現する予定である (図??)。現在のところ、core のシステム・コールの設計が終わった段階であり (付録参照)、core 上で実現されるサーバの機能については、ネットワーク透過でユーザ指向型なファイル・システムなど一部決まっているものを除いてこれから検討に入るところである。その詳細は別の機会に譲りたい。

謝辞

本分散 OS プロジェクトの世話をしていただき、いつも貴重な意見をいただいている京都大学数理解析研究所の中島玲二教授に深く感謝する。本プロジェクトに参



モジュールAのentry_1を呼び出したRPCの引数で渡ってきたケイパビリティをスレッドが持っているのは、1と5の間のみで3の間はremote_callの引数として再び渡されたものだけ(もちろんその中にentry_1の引数として渡されたものが入っていることはある)である。

また、モジュールBからの返り値にケイパビリティがあれば、5の間スレッドが持っていることになり、モジュールAからの返り値にならない限りentry_1を呼び出したところへは渡っていかない。

図7: スレッドのケイパビリティ

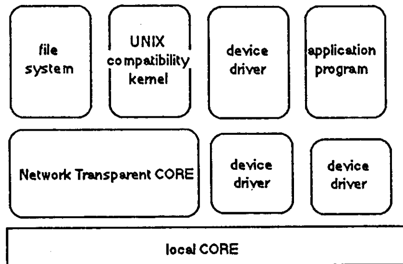


図8: ネットワーク透過 core

加し、活発に議論に参加していただいている企業(アステック、SRA、キャノン、ソニー、立石電機、東芝、日本アイビーエム、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、富士ゼロックス)の方々に感謝する。

参考文献

[1] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A. and Young, M.: "Mach: A New Kernal Foundation For UNIX Development", in *Proceedings of USENIX 1986 Summer Conference*, pp. 93–112, 1986.

[2] 新井 潤, 立木 秀樹, 荻野 達也, 服部 隆志, 森島 晃年: "分散OS ToMの構想 — その基本設計の概要 —", 京都大学数理解析研究所テクニカル・レポート RIMS-675, 1989.

[3] 新井 潤, 桜川 貴司, 立木 秀樹, 荻野 達也, 服部 隆志, 森島 晃年: "分散環境をサポートするOS ToMの構想 — そのプログラミング・モデルとセキュリティ機構 —", 日経エレクトロニクス, 10月16日号, pp. 187–199, 1989.

[4] Birrel, A. and Nelson, B.: "Implementing Remote Procedure Calls", ACM Transaction on Computer Systems, Vol. 2, No. 1, pp. 39-59, 1984.

[5] Cheriton, D.: "VMTP: A Transport Protocol for the Next Generation Communication Systems", Proceedings of the ACM SICOMM'86 Conference, Stowe, Vermont, August 5–7, 1986.

[6] Cheriton, D.: "The V Distributed System", CACM, Vol. 31, No. 3, pp. 314–333, 1988.

[7] Mullender, S. and Tanenbaum, A.: "The Design of a Capability-Based Distributed Operating System", The Computer Journal, Vol. 29, No. 4, pp. 289–299, 1986.

[8] Ousterhout, J., Cherenson, A., Douglis, F., Nelson, M. and Welch, B.: "The Sprite Network Operating System", IEEE Computer, Vol. 21, No. 2, pp. 23–36, 1988.

[9] Popek, G., Walker, B., Chow, J., Edwards, D., Kline, C., Rudisin, G. and Thiel, G.: "LOCUS: A Network Transparent, High Reliability Distributed System", IEEE Proc. Operating Systems Principle Symp., 1979.

[10] 寺岡 文男, 横手 靖彦, 所 真理雄: "オブジェクト指向分散オペレーティングシステムMUSEにおけるオブジェクト間通信機構", 日本ソフトウェア科学会第5会大会論文集, pp. 197–200, 1988.

[11] Leffer,S., McKusick,M., Karels,M. and Quarterman,J.: "The Design and Implementation of the 4.3BSD UNIX Operating System", Addison-Wesley, 1989.

付録: ToMのシステム・コール

ハウス

```
type house
house_create: () -> house
house_destroy: house * exit_status -> ()
```

ユーザ

```
type user
house_user: house -> user list
house_user_add: house * user list -> ()
house_user_delete: house * user list -> ()
```

モジュール

```
type module
module_create: house * bool -> module
module_house: module -> house
module_exclusive: module -> bool
module_destory: module -> ()
house_module_list: house -> module list
```

スレッド

```
type thread
thread_create: module * entry * argument list
                  -> thread
thread_real_house: thread -> house
thread_effective_house: thread -> house
thread_destroy: thread -> ()
thread_self: () -> thread
thread_module: thread -> module
thread_set_effective_house: thread * house
                  -> ()
house_thread_list: house -> thread list
house_foreign_thread_list: house
                  -> thread list
```

```
thread_sleep: queue * destroy_mode
                  -> wakeup_mode
thread_wakeup: queue -> ()
thread_suspend: thread -> ()
thread_continue: thread -> ()
thread_status: thread -> thread_status
thread_set_status: thread * thread_status
                  -> ()
```

エントリ

```
type entry_group
entry_register: module * entry
                * argument_type list
                   * argument_type list -> ()
module_entry_list: module -> entry list
entry_destroy: module * entry -> ()
entry_info: module * entry
   -> argument_type list * argument_type list
entry_group_create: module * entry list
                       -> entry_group
entry_group_destroy: module * entry_group
                       -> ()
entry_group_entry_list: module * entry_group
                       -> entry list
```

ケイパビリティ

```
capability_create: module * entry_group
                       -> capability
capability_module: capability -> module
capability_entry_group: capability
                       -> entry_group
capability_destroy: capability -> ()
house_capability_list: house
                       -> capability list
house_capability_add: house * capability list
                       -> ()
house_capability_delete:
                house * capability list -> ()
thread_capability_list:
                thread -> capability list
```

リモート・プロシジャ・コール

```
remote_call: capability * int * argument list
                   -> argument list
remote_return: argument list -> ()
```

ページング

```
type pager
pager_create: size * capability -> pager
pager_destroy: pager -> ()
pager_size_change: pager * size -> ()
pager_info: pager * region -> pager_info
pager_allocate: pager * region
                  -> pointer * pager_info
pager_deallocate: pointer * size -> ()
pager_flush: pager * region * flush_mode
                  -> ()
pager_copy: pager * region * pager * region
                  -> ()
module_set_pager: module * region
                     * pager * offset -> ()
module_pager_protect: module * region
                     * protection -> ()
module_pager_info: module * address
                    -> protection * pager
module_pager_map: module -> pager_map
```

エクセプション

```
type exception
exception_raise: thread * exception
                     * argument list -> ()
exception_catch: exception -> argument list
exception_catch_all:
            () -> exception * argument list
```