

『システム性能評価用の OS モデル』

柿元 満、高橋 敏哉、前田 誠司

(株) 東芝 研究開発センター 情報・通信システム研究所

概要

システム構築の現場では、後々の性能問題の発生を防ぐため事前に性能を予測するツールが要求されている。従来から、性能評価ツールとして、待行列網理論をベースにしたモデルによるシミュレーションが利用されているが、モデルの作成が必ずしも容易ではなかった。その一つの理由として、オペレーティングシステム（以下 OS）の動作のモデル化が困難であることがあげられる。OS の資源管理アルゴリズムが複雑であることに加えて、様々の OS 毎に逐一モデルを構築することが手間を要するからである。

本稿ではオブジェクト指向を利用して、OS のシミュレーションモデルを系統的に構築する試みを紹介する。様々な OS に共通に見い出せるメカニズムをベースクラスの手続ききとして組み込んでおき、各 OS に固有の動作はサブクラスの手続ききとして実装する。

Models of Operating System for Performance Evaluation

Mitsuru Kakimoto, Toshiya Takahasi, Seiji Maeda

Communication and Information System Laboratories, R&D Center, Toshiba co.

Evaluating performance at the early stage of system integration process is a efficient way to suppress potential performance related trouble. Though discrete event simulation of queueing network type models has been popular framework for performance evaluation, complicated resource management algorithms of operating systems, as well as their varieties, are major obstacles in describing system behaviors in terms of queueing network.

In this paper, we present a systematic method to build performance model of operating systems using object-oriented language. Characteristics commonly found in operating systems are modeled inside base classes, while those peculiar to individual operating systems are implemented in derived classes.

1 まえがき

顧客要求を満足するためにシステムエンジニア (SE) は様々なハードウェア機器やソフトウェアを組合せてシステムを構築しなければならない。このとき機能的な仕様を確認するのは比較的容易であるが、ハードウェアのスペックや OS、アプリケーションの動作特性を考慮して性能を定量的に見積もるのは容易ではない。安全を図るためあえて冗長なシステムを設計せざるをえない場合もある。これを解決するため設計段階のシステムを待行列網でモデル化し、シミュレーションによって性能が予測されている [1]。

このようなモデル化においては計算機システムは次の3つから構成されているとみなすことができる。

- 計算資源
プロセッサ、ディスクなどのハードウェア機器。ファイル、メッセージ、セマフォなどの論理的な資源も含まれる。
- 資源利用者
アプリケーションプログラムがこれに対応する。
- 資源管理アルゴリズム
OS がこれに対応する。

待行列網では、システムをサーバ（資源に相当する。窓口ともいう）とジョブ（資源利用者のこと。客ともいう）から構成されるとみなす。ハードウェア機器や OS の提供する資源をサーバ、ユーザの行なう処理の流れをジョブに対応させれば、おおまかには計算機システムのモデルが待行列網として記述できる。例えば、ディスク装置をサーバ、I/O 処理をジョブとみなせば複数の I/O 処理がディスクで競合する様を表現できるし、CPU をサーバ、その上で動くプロセスをジョブとみなせば、マルチタスクのモデルが待行列で表現できる。このようなモデルを用いてシミュレーションを行なった結果、あるサーバの待行列が長い、つまりサービス待ちのジョブが多いことが分かれば、そのサーバにクロックの速い CPU を用いて処理能力をあげたり、データを複数のディスクに分

割して負荷を分散させるなどして対処することができる。

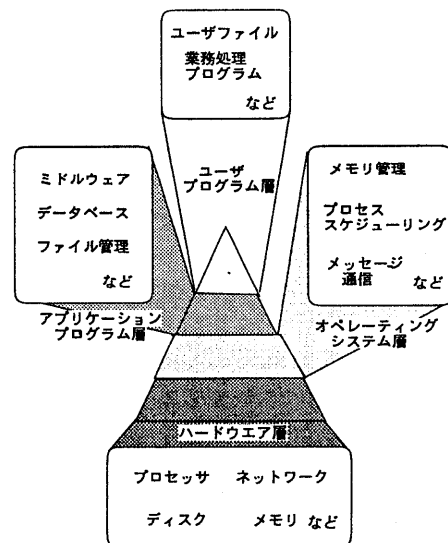


図 1: 計算機システムの階層性

ところで計算機システムは複数のハードウェア、ソフトウェアのモジュールが階層的に組合わさって構成されている。このため、計算機の動作全体を一人の SE が見通して、待行列のモデルとして書き下すのは難しい。一つの解決策として計算機システムの階層性、モジュール性に沿ったモデル構築を行なうことが考えられる。一般に計算機システムはハードウェア、OS、ミドルウェア、アプリケーションという階層構成を有している。個々の階層のモデルはそれに詳しい人に設計してもらい、それらを組み合わせて一つのシミュレーションモデルを構築するようなツールを提供できれば SE は個々のモデルの詳細に煩わされる必要がなくなる（図 1）。

この中で、OS は資源管理の中心であり、そのモデル化はもっとも重要な位置を占める。しかしながら、OS のモデルを作成する側にとって、世の中に存在する様々な OS のモデルを提供することは依然として大きな負担である。系統的にライブラリーの体系を作る手法を確立することが重要である。本稿ではオブジェ

クト指向を利用して、OS のシミュレーションモデルを系統的に構築する試みを紹介する。

2 OS モデル化の概要

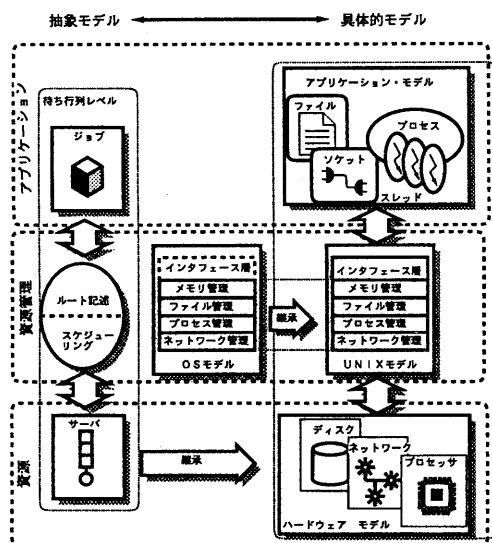


図2: OS モデルの構成

OS はプロセッサ、メモリ、ディスク、ネットワークなどのハードウェア資源の上にそれらを仮想化した論理的な資源（プロセス（スレッド）、ファイル、メッセージ）を構築している。扱うハードウェア資源は共通しているし、論理的な資源も多くの共通した構造を持っている。こうしたOS に共通してみられる構造を抽出したのがOS モデルで、次のサブモデルから構成される（図2）。

- (1) メモリ管理部モデル
- (2) プロセス（スレッド）管理部モデル
- (3) ファイルシステム部モデル
（バッファ管理、データ配置管理）
- (4) プロセス間通信部

各サブモデルはそれぞれメモリ管理、スケジューリング、ディスク上でのデータ配置、通信のアルゴリズムを記述する部分である。各サブモデルはそれぞれ C++ の一つのクラスとして定義されている。これらのクラスはそれが表現するサブモデルが持っている機能をシミュ

レートする手続きを持っている。

一方、個々の資源の管理方式は OS 固有の個性がある。またマイクロカーネルなどに象徴されるように、近年の OS はモジュールを組合せが柔軟になってきており、管理アルゴリズムが自由に設定できる度合いが大きくなっている。これに対応するため、ベースとなるモデルから個々の資源管理アルゴリズムの変更は柔軟にできなければならない。このために継承の概念を利用している。ベースモデルで定義されたクラスから、個々の OS の固有の資源管理アルゴリズムを組み込んだサブクラスを定義することにより特定の OS に対応したモデルを作ることができる。

また、OS モデルは外側に実際の OS と似たインタフェースを持っている。これによりアプリケーションの動作モデルを一般のプログラムと同じように記述できる。実際にある OS でのプログラミングに親しんだユーザにとっては大きな利点となる。特に既存のプログラムをシミュレーションモデルに変換する際に効果が期待できる。また、バージョンアップなどにより OS モデルの内部を変更した場合でも、インタフェースは変わらないのでアプリケーションモデルは継続して利用可能である。従って、資産としてのモデルの蓄積を計ることができる。

OS モデルを作るにあたってまず待ち行列網をシミュレートする C++ クラスライブラリを作成し、その上にベースモデル、更に特定の OS 用のモデルというように階層的に構築する方針にした。まず次節で待ち行列網ライブラリを説明し、続いて OS モデルの説明を行なう。

2.1 待ち行列網ライブラリ

待ち行列網ライブラリの一つの特長はプロセス指向を採り入れたことである。（図3）。プロセス指向は SIMULA で初めて採り入れられた考えかたで、Smalltalk80 のシミュレーションパッケージなどにも採用されている [1]。プロセス指向の“プロセス”とは、モデル内で並行に動作する処理の流れを抽象的に表現したものである。プロセス指向の現実的な利点

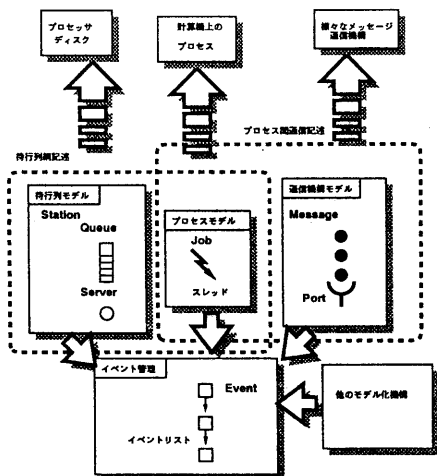


図 3: 待ち行列網ライブラリーの構成

はルート¹をプログラム言語の処理フローとして（簡単にいうと if then や for をもちいて）記述できることである。これにより、条件判断やループを含む複雑なルートの記述が比較的容易になり、複雑なアルゴリズムの記述に適している。この機構は OS（あるいはライブラリ）の提供するスレッドを使って実現している [6]。“プロセス”はモデル化対象の計算機システムのプロセスまたはスレッドに対応するようになっており、マルチタスクのアプリケーションのモデルを自然に記述できる。

計算機内には通信など、ジョブ間の待合わせや資源の動的な生成／消滅をとまなう動作が多くあり待行列だけのモデル化は難しい。このためプロセス間の通信に基づいたモデルも提供している。また、ユーザが独自のモデルを作れるように、シミュレーションのイベント管理を直接操作できるようにしている。

2.2 オブジェクト指向の枠組

先に述べた4つのサブモデルのうち、プロセス間通信部はまだ未完成である。ここでは残りの3つ

●メモリ管理部モデル

¹ジョブがまずどのサーバにはいつにどこへ…と
いうこと

- プロセス（スレッド）管理部モデル
- ファイルシステム部モデル

の実装について説明する。（なお、以下で述べるクラス定義はアイデアを明確にするため、説明用に実際のものとは多少変更してある）。

3つのサブモデルはそれぞれクラス MemoryManager, Dispatcher, FileSystem として、定義されている。またこの他に、OS の管理する資源であるメモリーのページ、スレッド、ファイルなどもそれぞれクラス Page, LWP, File として定義されている。

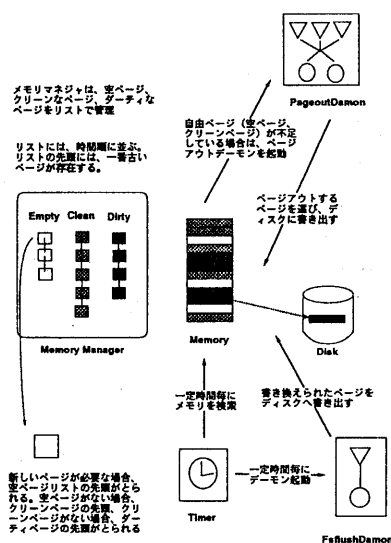


図 4: MemoryManager の構成

MemoryManager はページ単位の主記憶管理をシミュレートするオブジェクトである。主記憶はページ（そのサイズは任意に設定できる）の集合とみなされる。MemoryManager は“空”（内容が入っていない）“クリーン”（ディスクに反映済み）“ダーティ”（ディスクに未反映）の3種類のページのリストを保持している。

MemoryManager にはページアウトデモンを表すオブジェクト PageOutDaemon が備わっていて、“空”、“クリーン”なページが不足した場合に“ダーティ”なページをディスクに退避する。その場合、退避されるページの選択

```
virtual Page* get(LWP* lwp,
                  FileSystem* fs)
    自由なページのプールからページを
    一つ確保する。

void free(LWP* lwp, Page* page)
    ページを解放し自由ページとする。

void write(LWP* lwp, Page* page)
    ページの更新を通知する。

void clean(LWP* lwp, Page* page)
    ページがクリーンになったことを通
    知する。
```

図 5: MemoryManager の手続き (抜粋)

アルゴリズムはクラス PageOutDaemon に記述されているが、サブクラスを定義してそのアルゴリズムをオーバーロードすることができる。

MemoryManager は図 5 に示すようにページを取り出す／戻す手続きを持っており、これらは FileSystem などの他のサブモデルから呼び出される。

FileSystem は

- ディスク上のデータ配置
- ファイルバッファの管理

という性能上重要な 2 つの機能を持っている。バッファ管理はバッファのヒット率に大きく影響するし、データ配置管理は I/O の際、アームのシークに大きく影響する。これらの実装は OS 毎に大きく異なると考えられる。図 6 に FileSystem は手続きを示したが、この中で getPageLocation() と find() がそれぞれ機能を具体的に実現したアルゴリズムを記述する関数である。これらは仮想関数になっており、 subclasses においてオーバーロードできるようにしている。

クラス Dispatcher はプロセス (スレッド) のスケジューリングを司る。その機能は図 7 に

```
virtual void read(LWP* lwp,
                  File* file,int offset,int size)
virtual void write(LWP* lwp,
                   File* file,int offset,int size)
    ファイルの ( offset, size ) で指
    定された部分を読み込む。この 2 つ
    が FileSystem の主要な機能を表現
    した手続きである。

virtual struct location*
    getPageLocation( Page* page )
    データ配置部にデータのディスク上の
    所在を尋ねる手続き。

virtual Page* find( File* file,
                    int offset )
    バッファ管理部にデータの入っている
    主記憶上のページを尋ねる手続き。
```

図 6: FileSystem の手続き (抜粋)

```
void switch_in( LWP* lwp )
    LWP が動作可能になった時に呼び出さ
    れる手続き。これにより LWP が実行キ
    ューに積まれる。

void switch_out( LWP* lwp )
    LWP がブロックする時に呼び出される
    手 続 き。これにより、スケジューリ
    ング機能が起動され、次の LWP がプロ
    セッサ上で実行される。
```

図 7: Dispatcher の手続き (抜粋)

示した、二つの手続きを呼び出すことで起動される。

これらの手続きを階層的に組み合わせることによって、OSの機能をシミュレートできるようになる。例えば、データの読み込みを例にとってサブモデルがどのように動作するか見てみる。まず主記憶上のバッファに所望のデータがあるか（バッファ管理）調べられる。ない場合は、データを格納するためのメモリーを確保し、（メモリ管理）ディスク上でのデータの格納場所を求め（データ配置管理）ディスクからの読み込み動作が行なわれる。データを要求したプロセスはブロックし、他のプロセスが動かされる（プロセス管理）。このように読み込みと言う、マクロには一つの動作の中にも複数の管理アルゴリズムを起動する様を記述しなければならない。この処理の流れは

FileSystem::read()で実装されている（図8）。このような複数資源にまたがる一連の処理の流れは多くのOSで共通に見い出せる（これをシナリオと呼ぶことにする）。OSモデルにはこのような形で基本的なシナリオが組み込んである。

2.3 モデルの粒度

システムの性能に影響する現象はタイムスケールの小さいものから、大きいものまでさまざまなものが考えられる。細かい現象まで正確にシミュレートするモデルを作成すれば正確な性能予測を行なうことができるであろうが、そのために必要な計算コストも大きくなってしまふ。大規模なシステムをシミュレートするにはOSのすべての側面をシミュレートするモデルを作ることはできず、妥当な計算コストで妥当な結果を求められるように、現象を取捨選択しなければならぬ。従って、ここではマクロなレベルで性能に大きく寄与する現象のみをシミュレートし、ミクロなレベルの現象は無視する、または数値パラメータに繰り込んだ形で扱うこととした。

ここでマクロな現象とは次のようなものをいう。

- バッファのヒット / ミス

- プロセスの動作（実行／待ち合わせ／停止／コンテキストスイッチ）
- ディスクの動作（シーク、回転待ち）

一方ミクロな現象とは例えば次のようなものを指す。

- キャッシュのヒット／ミス
- TLBのヒット／ミス
- システム・バスの競合

モデルに完成させるためには各アルゴリズムを手続き的に記述するとともに、その実行時間をパラメータとして与える必要がある。当初はOSの動作を関数レベルに分割して、それぞれの関数の実行時間を計測してモデルに組み込む予定であった。しかしながら、その計測にはOSのソースコードが必要であり困難が伴う。現在はシステムコールのレベルで計る方法を模索中である。

2.4 OS 毎の相違への対応

性能評価のターゲットとなるシステムには様々な機器構成、OSを持った計算機が含まれる。OSモデルにも次の様なヴァリエーションに対応する必要が生じる。

- 同じOSでも走っているマシンが異なる。
 - プロセッサのクロックが異なる。
 - シングル or マルチ
 - プロセッサが異なる。
(Intel, Sparc, ...)
- バージョンが異なる。
- メーカーが異なる。
例えば UNIX でも Solaris, SVR4, AIX, BSD ... などいろいろある。
- OS が異なる。
UNIX, WindowsNT, OS/2, ...

理想的にはOSモデルはこれらの違いを吸収し、どのようなOSであっても妥当な制度で性能が予測できることが望ましい。私たちは以下のようなアプローチを採った。ある特定の

```

FileSystem::read( LWP* lwp, File* file, int offset, int size )
{
    読み込むデータをページ単位に分割;
    for( 各ページに対して ) {
        Page* page = find( file, ページの先頭のオフセット );
        if ( 主記憶上に存在しない ) {
            memoryManager->get( lwp, this ); // ページの確保
            ページ内容の設定;
            // データのディスク上の位置を求める
            struct location* loc = getPageLocation( page );
            物理 I/O の発行;
            dispatcher->switch_out( lwp ); // コンテキストスイッチ (アウト)
            物理 I/O の終了まで待つ;
            dispatcher->switch_in( lwp ); // コンテキストスイッチ (イン)
        }
    }
};

```

図 8: FileSystem::read() の詳細

マシン上で走る特定の OS を参照用 OS として、それに対して当てはめたモデルを作成する。他のマシンで走る他の OS のモデルは参照用 OS モデルのパラメータを変更する、あるいは部分的にコードを書き換えるなどの変更を加えることにより作成できると考えた。変更に関しては、いくつかのレベルを想定している。

- レベル 1 :
プロセッサスピードによる実行時間パラメータを校正する。校正には SPECInt で公表されたプロセッサスピードを用いる。
- レベル 2 :
実行時間パラメータを測定し直す。
- レベル 3 :
サブモデルのアルゴリズムを変更する。子クラスを定義することにより対応する。
- レベル 4 :
シナリオの変更を伴う改造を行なう。複数の子クラスを新たに作る必要がある。

同じ OS が異なったプロセッサ性能を持つマシンの上に載っている場合は、OS モデルは実行時間パラメータがプロセッサ性能に反比例して変更されるだけで十分であると期待される。これに対し、バージョンがことなる場合などはレベル 2 以上のに対応が必要になるだろうし、更に OS の構造自体が異なればレベル 3 以上の対応が必要になるであろう。これに関しては実証による検証が必要と考えている。

3 UNIX モデル

OS モデルの具体化の最初の実装例として UNIX を選んだ。これは、UNIX が今日最も普及している OS の一つであると同時に、メーカー独自の OS に比較して挙動の制御やチューニングが難しく、性能見積もりが困難であることが指摘されており、性能評価の需要が大きいと考えたからである。

UNIX にはプロセス、ファイル、ソケットなどのシステム資源があり、独自の資源管理アルゴリズムを持っている [2],[3],[4]。UNIX モデルにはこれらの資源管理アルゴリズムと、その実行に必要なプロセッサ使用時間がパラ

メータとして組み込まれている。

1993, pp. 33-44.

更に UNIX モデルのインタフェースでは UNIX のシステムリソースを表現するオブジェクトが C++ のクラスとして定義されている。アプリケーションの動作はこれらのオブジェクトに対する操作として記述できるので、UNIX 上でのプログラミングに親しんだユーザであれば、アプリケーションの動作モデルの記述も容易である。

4 あとがき

計算機システムの性能問題の中心となる OS のシミュレーションモデルの作成を容易にする手法の方針について説明した。現在は主なターゲットを UNIX に定めているが、今後は PC を始め、様々な OS がシステムに組み込まれると予想される。これらも含めて、本手法の有用性を検討をこれから行なってゆく予定である。

参考文献

- [1] M. Pidd(ed.), "Computer Modeling for Discrete Simulation", Wiley (1989)
- [2] M.J. Bach, "The Design of the UNIX Operating System", Prentice Hall(1986), (坂本文、多田 好克、村井 純 訳 「UNIX カーネルの設計」、共立出版 (1991))
- [3] S. J. Leffer, M.K. Mckusick, M.J. Karels & J.S Quaterman, "The Design and Implementation of 4.3BSD UNIX Operating System", Addison-Wesley (1989) (中村 明、相田 仁、計 宇生、小池 汎平 共訳、「UNIX 4.3BSD の設計と実装」丸善 1991)
- [4] B. Goodheart & J. Cox, "Magic Garden Explained", Prentice Hall, 1994
- [5] H. Custer, "INSIDE WINDOWS NT", Microsoft Press (1993) (鈴木 慎司 監訳、福崎 俊博 訳 「INSIDE WINDOWS NT」アスキー出版局 (1993))
- [6] J. Sang et al. "Design and Implementation of a Simulation Library using Lightweight Processes", 1993 Summer USENIX, June,