

*Tender*におけるOS動作の可視化に向けた 情報収集機構の実現

河原 太介† 谷口 秀夫††

†岡山大学大学院自然科学研究科

††岡山大学工学部

OS動作に関する情報が容易に得られれば、オペレーティングシステムの改版や部分開発を効率よく行うことが可能となる。これを支援する具体的な手法のひとつとしてOS動作の可視化がある。我々はこれまでに、OS動作の可視化に必要な情報が容易に取得できるOSのプログラム構造を提案した。また、そのようなプログラム構造を有するオペレーティングシステム *Tender* における可視化の実現方法を提案した。本稿では、これらの研究に基づき、OS動作の可視化に向けた情報収集機構を *Tender* に実現するにあたっての課題と対処について述べる。また、実現した機構によって収集した情報をもとに、簡単な可視化の例を示し、機構の評価を行う。

Implementation of the information gathering mechanism towards visualization of behavior of the operating system in *Tender*

Taisuke KAWAHARA † Hideo TANIGUCHI ††

†The Graduate School of Natural Science and Technology, Okayama University

††Faculty of Engineering, Okayama University

If the information about operating system behavior is acquired easily, it become possible to perform the revision and partial development of an operating system efficiently. Visualization of operating system behavior is one of the methods for it. We proposed the program structure of operating system which information required for visualization of operating system behavior can acquire easily. Moreover, the implementation method of the visualization in operating system *Tender* which has such program structure was proposed. In this paper, we describe issues and managements for implementation of the information gathering mechanism towards visualization of behavior of the operating system in *Tender* based on these researches. And, we show the example of simple visualization using the information collected according to the implemented mechanism, and evaluate a mechanism.

1 はじめに

近年の目覚ましいハードウェア性能の向上は、大きな処理負荷を必要とする応用プログラム(以降、APと略す)の実行を可能にしている。また、計算機の価格性能比の向上により、計算機が一層普及するに従い、その利用形態も多様化している。このような背景に基づき、APの動作を制御するオペレーティングシステム(以降、OSと略す)に対し、その機能の高度化が求められている。

この要求に応えるため、OSは機能拡充を目的と

する改版が何度も行われる。また、新しいOSの開発も行われている。その際、開発コスト軽減のために既存のOSを部分的に利用することがある。このとき、OSの動作に関する情報、例えばOS内部モジュールの呼出関係や処理内容の情報を得ることができれば、開発の工数を削減することができる。これを支援する具体的な手法の一つとして、OS動作情報の可視化がある。なお、OSの学習を支援することを目的とした可視化の研究も行われている[1]。

我々は、これまでにOS動作情報の可視化に必要

な情報が容易に取得できる OS のプログラム構造を提案した [2]。また、そのようなプログラム構造を有するオペレーティングシステム *Tender*[3]を開発し、可視化の実現方式を提案した [4]。

ここでは、文献 [2]～[4]に基づき、*Tender*の OS 動作情報の可視化に必要な情報を収集する機構について、実現時の具体的な課題と対処を述べる。また、実現した機構によって得られる情報の例を示す。最後に、*Tender*が AP に提供する機能インタフェース（以降、カーネルコールと呼ぶ）について、本機構が与えるオーバーヘッドを計測し、本機構の評価結果を報告する。

2 *Tender* オペレーティングシステム

2.1 資源の分離と独立

Tender では、OS が操作する対象を資源として分離し、独立化している。資源には、資源識別子と資源名を付与し、資源操作のインタフェースを統一している。各資源を操作するプログラム部品を資源ごとに分離し、共有プログラムを排除している。また、各資源の管理情報も資源ごとに分離している。

2.2 プログラム構造

Tender のプログラムは、基盤部・表プログラム構造部・入出力制御部・拡張部の 4 つの部分で構成される。資源を管理する OS プログラムは、表プログラム構造部で管理する。

表プログラム構造では、OS プログラムをプログラム部品と呼ぶ単位に分割し、独立化する。また、各プログラム部品の呼出は、“操作対象の資源の種類”と“操作の種類”から構成されるプログラム呼出表を使って行う。このプログラム呼出表を図 1 に示す。“操作対象の資源の種類”はプロセスや実メモリといった資源の種類であり、“操作の種類”は OPEN, CLOSE, READ, WRITE, CONTROL のいずれかである。ここでは、このプログラム部品により行われる処理を資源処理と呼び、情報収集の対象とする。

3 基本方式

3.1 基本設計

ここでは、可視化機能としての処理を以下の 4 段階で構成する。

- (1) 可視化情報の収集 (収集部)
- (2) 可視化情報の受け渡し (転送部)

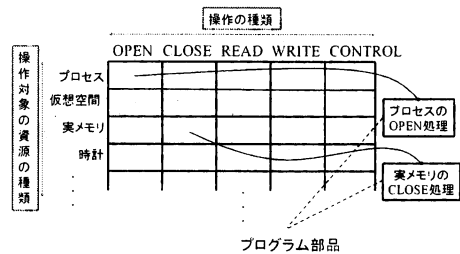


図 1 プログラム呼出表

- (3) 可視化情報を元にした OS の振る舞いの解析 (解析部)

- (4) 解析データの可視化表示 (表示部)

収集部は、OS のデータ構造や処理構造と密接に関連し、情報収集の対象となる資源処理を常に監視している必要がある。収集部が OS 外部にあると、OS 内部の処理とユーザ処理間の切り替えが多発し、OS 全体としての性能に無視できない影響を与えかねない。そこで、収集部は OS 内部の機能の一つとして実現する。

解析部と表示部では、データ処理や画面表示の処理が頻繁に行われる。このため、この 2 つの処理部を OS 内部に実現すると、OS 内部で負荷の大きな処理が多発するため、OS 全体としての性能が低下する。これを避けるため、この 2 つの処理部は OS 外部に実現する。

以上のことから、OS 内部の収集部と OS 外部の解析部の間で可視化情報を受け渡しするため、転送部を設ける。

3.2 収集する情報

OS 動作の情報を可視化するために、以下の情報を資源処理ごとに記録する。

- (1) 処理の要求元を示す情報
- (2) 処理の要求先を示す情報
- (3) 処理内容
- (4) 処理結果
- (5) 処理を要求したプロセスの情報
- (6) 処理の行われた時刻

本機構では、表 1 に示す項目を記録することにより、上記の情報の収集を行う。dest_rid は、処理の要求先を示す情報であり、処理の対象となった資源の資源識別子を記録する。operate は、処理内容に関する情報であり、OPEN, CLOSE, READ, WRITE, CONTROL のいずれかを意味する情報を記録する。

表 1 記録する項目

項目名	内容
dest_rid	操作対象の資源識別子
operate	操作の種類
ret_val	戻り値
pid	プロセス識別子
call_k_time	呼出時のカーネル時刻
ret_k_time	復帰時のカーネル時刻

ret_val は、処理結果に関する情報であり、処理の戻り値を 32 ビット符号無し整数として記録する。pid は、処理を要求したプロセスの情報であり、処理を要求したプロセスのプロセス識別子を記録する。call_k_time は処理を開始した時のカーネル時刻であり、ret_k_time は処理が終了した時のカーネル時刻である。

これらの情報を記録するために、動作記録表と呼ぶデータ構造を OS 内に用意する。動作記録表は、表 1 の項目を 1 エントリとする配列形式のデータ構造である。従って、動作記録表の 1 エントリの記録が、一回の資源処理の情報に相当する。また、以後動作記録表の各エントリ番号を、そのエントリに対応する資源処理の処理番号と呼ぶ。各資源処理は処理番号によって一意に識別される。

ここで、表 1 には処理の要求元を示す情報に相当する項目が無い。各処理の要求元を示す情報は、呼出元の処理の記録エントリを指す情報、すなわち呼出元の処理番号であるといえる。しかし、この情報は解析部において記録済みのエントリを検索することで得られるため、項目として記録しない。詳細については 4.2.2 節で述べる。

処理の行われた時刻については、これまでに可変な時刻進捗を持つ資源「時計」[5]を用いる方法を提案した。しかし、現在の *Tender* ver9.1 の時点では、資源「時計」により得られる時刻の精度が十分ではない。よって、ここでは、資源「時計」の代わりにハードウェアクロックカウンタを用いて、CPU 起動時からの動作クロック数をカーネル時刻として記録する。

3.3 情報収集機構の構造

Tender では、資源処理の要求は、全て RIC を介して行う。RIC は、資源処理の要求があると資源処理呼出モジュールを呼び出す。資源処理呼出モジュールでは、呼出表を参照し、要求された“資源

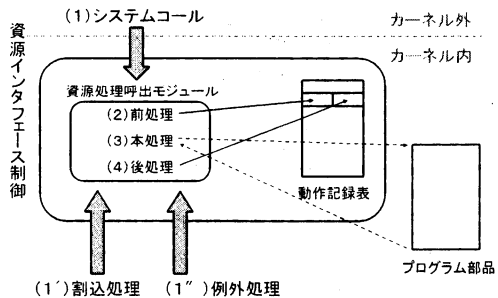


図 2 情報収集機構の概要

の種類”と”操作の種類”の組み合わせに対応するプログラム部品を呼び出し、制御を移す。

そこで、この資源処理呼出モジュールが各プログラム部品を呼び出す前と呼び出した後に、必要な情報を記録するように資源処理呼出モジュールの機能を拡張することによって、OS 動作の可視化に向けた情報収集機構を実現する。このようにすることで、全ての資源処理に対して情報の収集を行うことができる。この方式による情報収集機構の概要を図 2 に示す。ここで、呼出表を参照し、要求された各プログラム部品を呼び出す処理を本処理、本処理の前に行う記録処理を前処理、本処理の後に行う記録処理を後処理と呼ぶ。前処理時に記録される項目は表 1 の dest_rid, operate, pid, call_k_time である。これらの情報は、動作記録表の先頭のエントリから順に書き込まれる。後処理時に記録される項目は、ret_val, ret_k_time である。これらの項目は、当該処理の前処理時に記録が行われたエントリに書き込まれる。

dest_rid, operate, pid は、処理の呼出元から前処理へパラメータとして渡される。ret_val は、本処理の戻り値が後処理へパラメータとして渡される。call_k_time, ret_k_time は、それぞれ前処理と後処理において、CPU 起動時からの動作クロック数をハードウェアクロックカウンタによって取得する。

4 課題と対処

4.1 課題

OS 動作情報の可視化に向けた情報収集機構の実現には、以下の三つの課題がある。

(課題 1) 資源生成処理における資源識別子の記録

表 1 で示したとおり、本機構では操作対象の資源識別子を dest_rid として記録する。また、この項目

表 2 図 3 に対応する動作記録表

インデックス	処理内容	前処理	後処理
1	処理 A	(1)	(6)
2	処理 B	(2)	(5)
3	処理 C	(3)	(4)

は前処理で記録する。各資源処理は、RIC に操作対象の資源識別子と操作の種類をパラメータとして指定することで要求される。そこで、dest_rid の項目はそのパラメータを用いて記録する。しかし、5 種類ある操作の中で、OPEN 操作すなわち資源の生成処理にあたる資源処理のみ、RIC インタフェース関数に、パラメータとして“操作対象の資源識別子”と“操作の種類”ではなく、“操作対象の資源の種類”と“操作の種類”を指定して要求される。そして、生成処理の戻り値として、その処理で生成された資源の資源識別子が得られる。これが OPEN 操作における“操作対象の資源識別子”に相当する。よって、OPEN 操作においては、他の操作のように前処理で dest_rid を記録することはできないため、何らかの対処が必要となる。

(課題 2) 処理の要求元を示す情報の取得

3.2 節で述べたとおり、処理の呼出関係を可視化するためには、各処理の要求元を示す情報が必要となる。しかし、処理の要求元を示す情報は、処理の要求先を示す情報のように呼出元からパラメータとして与えられない。そのため、この情報を取得するための方法を検討する必要がある。

(課題 3) 後処理における適切な記録エントリの特

定
資源処理が内部でさらに他の資源処理を呼び出すような場合、記録が入れ子となる。この例を図 3 に示す。図中の矢印は処理が移ることを意味する。また、動作記録表への書き込みはこのとき行われ、矢印の数字は処理の移る順番であり、かつ記録処理が行われた順番を意味する。この処理の記録に対応する動作記録表の概要を表 2 に示す。表中の数字は、図 3 中の数字に対応し、いつそのエントリに対して記録処理が行われるかを意味している。この図の一連の資源処理において、処理 B の前処理は (2) のとき行われる。そして、処理 C の前処理 (3) と処理 C の後処理 (4) の後、処理 B の後処理 (5) が行われる。このように、一つの資源処理の前処理と後処理は必ずしも続けて行われるとは限らない。従って、後処理では毎回何らかの方法で、前処理で記録が行われたエントリを特定する必要がある。

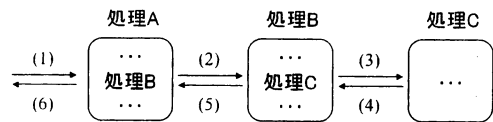


図 3 呼出が入れ子になる資源処理の例

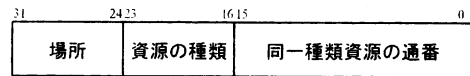


図 4 資源識別子の構成

4.2 対処

4.2.1 資源生成処理における資源識別子の記録法

この課題への対処を述べる前に、まず資源識別子について説明する。資源識別子は 32 ビットの符号無し整数で表現され、その構成は図 4 のようになっている。資源識別子は、分散環境における場所情報、資源の種類、同一資源種類の通番からなる。つまり、資源識別子は資源の種類の情報を包含している。従って、この課題への対処として、次の二通りの方法が考えられる。

(対処 1) OPEN 操作に対応する資源処理の記録時のみ、前処理では完全な“資源識別子”ではなく“資源の種類”の部分のみを dest_rid として記録する。そして、後処理で“同一種類資源の通番”部分を処理の戻り値より補完し、dest_rid を“操作対象の資源識別子”の記録として改めて記録する。

(対処 2) “操作の種類”に関わらず、全ての資源処理において、dest_rid の項目は後処理で記録する。

(対処 2) は、すべての処理に対して統一的な処理を行うことによって、機構がシンプルになるというメリットがある。しかし、後処理が終わっていないすべての処理について、操作の対象が不明となる。これは、リアルタイムに可視化を行うことを想定した場合、見過ごせないデメリットである。従って、(対処 1) を採用する。

4.2.2 処理の要求元を示す情報の取得法

処理の要求元を示す情報、すなわち呼出元の処理番号を取得する方法として、次のような方法が考えられる。

(対処 1) 前処理において、呼出元の処理番号を動作記録表に記録する項目の一つとして記録する。

表 4 (課題 2) の対処の比較

	長所	短所
(対処 1-1)	処理負荷が小さい	資源処理呼出のインタフェースが煩雑になる。
(対処 1-2)	高い精度の時刻を必要としない。	OS 内部の処理負荷が大きい
(対処 2)	OS 内部の処理負荷が小さい。	(1) OS 外部の処理負荷が大きい。 (2) 高い精度の時刻が必要である。

表 3 図 5 に対応する動作記録表

インデックス	処理内容	前処理	後処理
1	処理 A	(1)	(6)
2	処理 B	(2)	(3)
3	処理 C	(4)	(5)

(対処 2) 解析部において、既に記録してある内容から、各処理の呼出元の処理番号を検索する。

(対処 1) は、さらに二通りの方法が考えられる。

(対処 1-1) 資源処理呼出の際に、自分の処理番号を渡すようにすべての資源処理のインタフェースを拡張する。

(対処 1-2) 前処理において、既に記録してある内容から、各処理の呼出元の処理番号を検索する。

(対処 1-1) では、資源処理呼出の際に RIC へ当該処理の処理番号を渡すように、すべての資源処理のインタフェースを拡張する。そして、前処理時に呼出元から渡された処理番号を、呼出元の処理を示す情報として記録する。

(対処 1-2) では、前処理時に、その時点で既に記録が行われているエントリの中から、呼出元の処理のエントリを検索する。しかし、図 5 の例のように、直前に記録が行われた処理が呼出元の処理であるとは限らない。図 5 の例では、処理 A が処理 B を呼び出し、処理 B の終了後、処理 A が処理 C を呼び出している。(4) の時点で、直前に記録が行われた処理は処理 B であるが、処理 C の呼出元は処理 A である。図 5 の一連の処理を記録した動作記録表の概要を表 3 に示す。そこで、以下の条件を最初に満たすエントリを、当該処理の記録エントリからさかのぼって検索する必要がある。

(条件 1) 後処理がまだ行われていない。

(条件 2) そのエントリの pid と、当該処理の pid が一致する。

続いて (対処 2) を説明する。記録した時刻の精度が十分であれば、(対処 1-2) の (条件 1) を、解析部において以下の (条件 1) のように言い換えることができる。そこで、この方法では解析部において、(対処 1-2) とほぼ同様の手続きを記録済みのエントリ

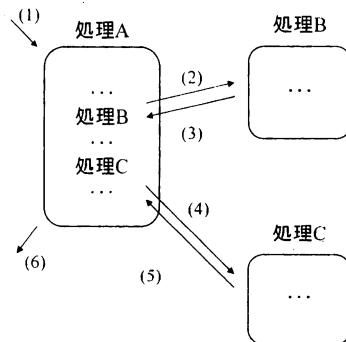


図 5 直前に行われた処理が呼出元の処理ではない例

それぞれに対して行う。具体的には、記録済みの各エントリに対して、以下の条件を最初に満たすエントリを、当該処理の記録エントリからさかのぼって検索する。

(条件 1) 自分の call.k_time よりもそのエントリの ret.k_time のほうが大きい。

(条件 2) そのエントリの pid と、その処理の pid が一致する。

これらの対処の比較を表 4 に示す。

(対処 1-1) では、全ての資源処理呼出のインタフェースに、処理番号を渡すためのパラメータを追加する必要がある。そのため、資源処理呼出のインタフェースが煩雑になる。一方、呼出元の処理を示す情報を記録するためには、呼出元から与えられる処理番号を記録するだけで済むため、検索処理を必要とする (対処 1-2) と対処 2 に比べて処理負荷は小さい。(対処 1-2) では OS 内部の処理である前処理で検索処理を行うため、他の対処に比べて OS 内部での処理負荷が大きい。一方、(対処 2) では解析部で検索処理を行うため、他の対処に比べて OS 外部での処理負荷は大きい。また、(対処 2) によって正確に呼出元のエントリを特定するためには、各エントリの前処理と後処理が行われた順番を明確に区別するために十分な精度で各カーネル時刻が記録されている必要がある。一方、(対処 1-2) では時刻の精度に

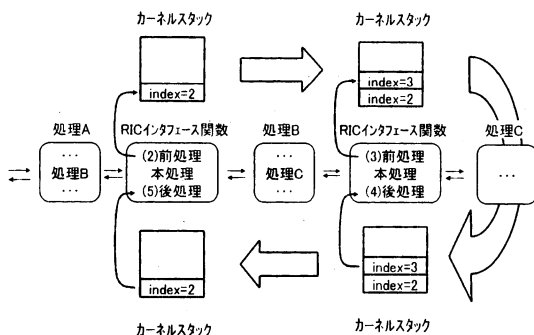


図 6 カーネルスタックによる記録位置の特定処理

依存することなく呼出元の検索処理を行うため、高い精度の時刻を必要としない。

(対処 1-1) について、処理負荷が小さいことを差し引いても、資源処理呼出のインタフェースが煩雑になるのは見過ごせないデメリットである。次に(対処 1-2) と (対処 2) について比較する。同じだけの処理負荷が、(対処 1-2) では OS 内部にかかるのに対し、(対処 2) では OS 外部にかかる。OS 内部の処理負荷はできるかぎり少ないほうがよいので、(対処 2) のほうが望ましい。また、CPU 起動時からの動作クロック数をカーネル時刻として記録する方法を採用しているため、(対処 2) のための精度は十分である。従って、(対処 2) を採用する。

4.3 後処理における適切な記録エントリの特定法

この課題への対処としては以下の二通りの方法が考えられる。

(対処 1) 以下の手順で行う。

- (1) 前処理で記録を行ったエントリの位置を、本処理呼出前にカーネルスタックに退避しておく。
- (2) 本処理終了後に、前処理で記録を行ったエントリの位置をカーネルスタックより復帰して、後処理に渡す。

(対処 2) 後処理の時点で、以下の条件を最初に満たすエントリを古いほうから検索する。

- (1) 後処理がまだ行われていない。
- (2) そのエントリの pid と、当該処理の pid が一致する。

図 3 の処理 B と処理 C の記録を、対処 1 により行う様子を図 6 に示し、以下に説明する。また、この処理に対応する動作記録表は表 2 である。

表 5 rsckind と資源の種類の対応

rsckind	資源の種類
VMEM	仮想空間
VREG	仮想領域
VKM	仮想カーネル空間
PMEM	実メモリ

処理 A 中で、RIC に対して処理 B が要求されると、まず (2) の処理 B の前処理が行われる。この処理では、動作記録表の 2 番目のエントリに記録が行われ、そのエントリ番号がカーネルスタックに積まれる。次に本処理として処理 B が呼び出される。同様にして、処理 B 中で RIC に対して処理 C が要求されると、(3) の処理 C の前処理が行われ、続いて処理 C が呼び出される。

処理 C が終了すると、RIC インタフェース関数は (4) の処理 C の後処理を呼び出す。その際、(3) の処理 C の前処理が行われたエントリ番号 3 をカーネルスタックより復帰し、(4) の処理 C の後処理に渡す。(4) の処理 C の後処理では、動作記録表の 3 番目のエントリに記録を行う。処理 B が終了すると、(4) と同様にして、(5) の処理 B の後処理が行われる。

(対処 2) では、後処理の時点で前述の条件を満たすエントリが、その処理の前処理が行われたエントリであることに基づき、検索処理によって適切な書き込み位置を特定する。

(対処 2) は、対処 1 に比べて記録済みエントリの検索を行うために、OS 内部でより多くの処理負荷が発生する。そこで、本機構では対処 1 を採用する。

5 動作例

ここでは、実装した情報収集機構の動作例として、資源「仮想空間」の生成処理の記録を取り、簡単な可視化の例を示す。

一連の処理を記録した動作記録表の例を表 6 に示す。各行が一つのエントリに対応する。dest_rid, ret_val, pid, call_k.time, ret_k.time の値は 16 進表現である。rsckind は操作の対象となった資源の種類であり、この情報は dest_rid より抽出できる。この rsckind の値と資源の種類の対応を表 5 に示す。また、src.index は呼出元の処理番号であり、動作記録表の他の項目から 4.2.2 項で述べた方法により取得し、記入した。

次に、簡単な可視化の例として、この動作記録表を元にこれらの処理の呼出関係を解析し、呼出関係を

表 6 資源「仮想空間」の生成処理に対応する動作記録表の例

index	src_index	dest_rid	rsc_kind	op_type	ret_val	pid	call_k.time	ret_k.time
1	-1	d0007	VMEM	OPEN	7	a0007	5b1b8356	5b205fa4
2	1	b0093	VREG	OPEN	93	a0007	5b1ba794	5b1c3462
3	2	2124b	PMEM	OPEN	143b	a0007	5b1c2dea	5b1c2fee
4	1	3007d	VKM	OPEN	7d	a0007	5b1c3fd7	5b1dc370
5	4	b0093	VREG	CONTROL	1000	a0007	5b1c4eae	5b1c507a
6	4	d0001	VMEM	CONTROL	0	a0007	5b1c52d1	5b1dc908
7	6	b0093	VREG	CONTROL	1000	a0007	5b1c5825	5b1c592e
8	6	b0093	VREG	CONTROL	143b000	a0007	5b1dc252	5b1dc38f

```

-----pid: a0007 -----
1:VMEM:OPEN (d0007:7) [708]
2:VREG:OPEN (b0093:93) [80]
3:PMEM:OPEN (2143b:143b) [1]
4:VKM:OPEN (3007d:7d) [225]
5:VREG:CONTROL (b0093:1000) [1]
6:VMEM:CONTROL (d0001:0) [213]
7:VREG:CONTROL (b0093:1000) [1]
8:VREG:CONTROL (b0093:143b000) [1]

```

図 7 呼出関係のツリー

に従ってネストしたものを図 7 に示す。

図 7 は、処理番号 1 の処理が開始してから終了するまでに呼び出される全ての処理の記録を、呼出関係に従ってネストし、ツリー状に表示したものである。この図において、処理番号 1 の処理が呼び出している処理は、処理番号 2、4 の処理であり、処理番号 2 の処理が呼び出している処理は、処理番号 3 の処理である。

図中の各ノードは、以下の形式で表されている。

```
処理番号:rsckind:operate(dest_rid:ret_val)[実行時間]
```

実行時間は、処理が開始してから終了するまでの時間(μ秒)である。これは ret_k.time と call_k.time の値、CPU の動作周波数により求まる。

6 評価と考察

実現した機構の評価として、実現した機構が各資源処理に与えるオーバーヘッドを計測した。具体的には、**Tender** が AP に提供する 61 種類のカーネルコールについて、情報収集を行っているときの実行時間と行っていないときの実行時間を測定した。測定には、PentiumII 450MHz の計算機を用いた。測定カウンタにはハードウェアクロックカウンタを使用した。処理時間の測定は、同じ処理を 50 回繰り返し、1 回あたりの処理時間を求めた。ディスク I/O が発生するカーネルコールについては、記録処理に

よる影響が明確にならないと判断し、測定の対象外とした。

表 7 に測定結果を示す。増加率は次式で求めた。

$$(\text{増加率}) = \frac{(\text{記録時}) - (\text{通常時})}{(\text{通常時})} \times 100$$

*1 は、各カーネルコールが終了するまでに行われた資源処理の回数である。この値は、収集した情報を解析することにより求めた。*2 は、各カーネルコール中で行われた資源処理 1 回あたりのオーバーヘッドであり、次式で求めた。

$$(*2) = \frac{(\text{記録時}) - (\text{通常時})}{(*1)}$$

表 7 から次のことがわかる。

- (1) カーネルコール呼出 1 回あたりにかかるオーバーヘッドは、カーネルコール中で行われる資源処理の回数が多いほど大きくなる。しかし、そのようなカーネルコールほど通常時の実行時間が大きい傾向があるため、増加率も大きいとは限らない。
- (2) *2 の値の平均は 0.9μ秒であった。つまり、資源処理 1 回あたりのオーバーヘッドは平均で 0.9μ秒であると言える。

7 おわりに

OS 動作の可視化機能を **Tender** に実現するための具体的な方式について述べた。また、提案方式に基づき、情報収集機構を実現した。この機構の実現には、資源生成処理における資源識別子の記録、処理の要求元を示す情報の取得、および後処理における適切な記録エントリの特定制、といった 3 つの課題があることを示し、それぞれについて対処を述べた。また、実現した機構により実際に資源処理の記録を取り、簡単な可視化結果を示した。さらに、実現し

た機構の評価として、カーネルコールについて可視化のための処理オーバーヘッドを示した。

残された課題としては、資源「時計」によるプロセスごとの時刻の進度を考慮した時刻の記録、実現した機構の評価がある。

参考文献

- [1] 西野洋介, 早川栄一, 高橋延匡, “可視化によるOS教育支援環境の実現と評価,” 情報処理学会研究報告, 2001-OS-88, pp.107-114(2001).
- [2] 野口裕介, 後藤真孝, 谷口秀夫, 牛島和夫, “OS動作の可視化機能,” 電子情報通信学会技術研究報告 Vol.96, No.33, pp.55-60(1996).
- [3] 谷口秀夫, “分散指向永続オペレーティングシステム *Tender*,” 情報処理学会シンポジウム論文集, Vol.95, No.7, pp.47-54(1995).
- [4] 野口裕介, 谷口秀夫, 牛島和夫, “OS動作の可視化機能の設計,” 情報処理学会シンポジウム論文集, Vol.96, No.7, pp.139-146(1996).
- [5] 野口裕介, 後藤真孝, 谷口秀夫, 牛島和夫, “*Tender*における資源「時計」の実現” 情報処理学会第52回全国大会, 4M-8, pp.43-44(1996).

表7 カーネルコールの実行時間 (μ 秒)

通番	通常時	記録時	増加率 (%)	* 1	* 2
1	17.7	18.9	6.8	1	1.2
2	11.7	12.5	6.9	1	0.8
3	11.3	12.4	9.6	1	1.1
4	11.5	12.3	6.7	1	0.8
5	1.9	2.7	38.8	1	0.8
6	456.4	491.8	7.8	39	0.9
7	16.0	20.5	28.3	5	0.9
8	45.2	67.5	49.4	24	0.9
9	51.7	73.2	41.5	24	0.9
10	39.5	60.2	52.5	22	0.9
11	1062.9	1128.7	6.2	71	0.9
12	79.5	103.1	29.6	22	1.1
13	7.3	8.0	10.2	1	0.7
14	1.8	2.6	40.7	1	0.7
15	1.6	2.4	48.1	1	0.8
16	1.8	2.5	40.4	1	0.7
17	1.5	2.2	44.5	1	0.7
18	1.7	2.5	44.1	1	0.8
19	14.1	15.0	6.0	1	0.9
20	1.5	2.3	51.5	1	0.8
21	1.7	2.4	42.1	1	0.7
22	14.7	15.6	5.7	1	0.8
23	1.6	2.4	45.9	1	0.8
24	1.5	2.4	57.2	1	0.9
25	1.6	2.3	42.4	1	0.7
26	1.7	2.5	42.3	1	0.7
27	14.4	15.8	9.5	1	1.4
28	7.8	9.3	18.6	2	0.7
29	6.1	6.9	12.7	1	0.8
30	6.5	7.9	20.9	2	0.7
31	5.6	6.4	13.8	1	0.8
32	19.3	22.0	14.2	3	0.9
33	22.4	26.0	15.9	3	1.2
34	24.6	34.5	40.2	10	1.0
35	3.2	4.2	29.6	1	1.0
36	4.6	5.5	18.3	1	0.8
37	1.8	2.7	48.1	1	0.9
38	1939.5	2069.1	6.7	90	1.4
39	16.5	24.0	45.9	9	0.8
40	300.3	350.6	16.8	45	1.1
41	1567.8	1660.8	5.9	67	1.4
42	19.8	23.0	15.9	1	3.1
43	9.5	10.6	12.2	1	1.2
44	2.1	3.0	43.1	1	0.9
45	681.3	690.5	1.3	8	1.1
46	2.1	2.7	30.0	1	0.6
47	344.4	354.6	3.0	11	0.9
48	4.0	4.9	21.7	1	0.9
49	5.1	5.9	15.5	1	0.8
50	98.9	108.0	9.3	9	1.0
51	15.0	15.7	4.7	1	0.7
52	2.4	3.2	33.3	1	0.8
53	1.8	2.6	43.8	1	0.8
54	1.6	2.4	46.1	1	0.8
55	10.0	11.6	16.0	2	0.8
56	3.0	4.4	48.0	2	0.7
57	22.8	24.4	7.1	2	0.8
58	2.8	4.3	52.9	2	0.7
59	28.5	30.0	5.1	1	1.1
60	19.1	23.1	21.2	13	0.3
61	6.0	7.9	31.6	2	0.9
平均	-	-	-	-	0.9