Linux 用ソフトウエア・イベント・トレーサ

堀川　隆
NEC ネットワークス　ネットワークス開発研究所

あらまし

プロセス切り換えや disk アクセスといったソフトウエア・イベントをトレースするイベント・トレーサを開発した。本稿では、そのトレーサの設計（測定対象イベントの選定)、実装方式（Linux kernel への組み込み方法)、適用例（性能分析の一例）を述べる。設計はイベント・トレースを系統立てて行なうために提案したフレームワークに基づいて行ない、実装は Linux kernel 内に hook を設置して必要なソフトウエア・イベントを検出する方式を採用した。イベント・トレース・ベース性能分析の効果を端的に示す例として、2.2 系の Linux kernel を使用するシステムにおいて複数のファイルを並行してアクセスした場合に発生する性能劣化現象を解明した事例を示す。

１．はじめに

一般に、どのような技術（engineering）であっても、１）対象の測定、２）対象のモデル化、３）対象の制御（測定とモデルをベースとする)、が基本である。ソフトウエア開発も例外ではなく、例えば、機能設計した通りに動かないプログラムを修正する、というデバグ作業をとってみても、測定はプログラムの状態（各変数の値や実行している命令の位置）を調べること、モデル化は（バグを発見するために）測定結果からプログラムの動作を推定すること、制御はバグを修正すること、と対応させることができる。

ソフトウエア開発では、機能と同様、性能も完成品（プログラム）に作り込む必要がある[1]。「性能を作り込む」とは、完成したプログラムが実用に耐える（理想的には設計した通りの）性能を発揮することを意味する。機能を作りこむ過程でデバグ作業が必要になることがあるのと同様、性能を作り込む過程でも性能的なデバグ作業（以下、性能デバグと略す）が必要となる場合がある。性能がそれほど大きな問題とならない場合は単にプログラムを動作させてその性能を確認するだけで済んでしまう反面、一旦、性能が問題になると、その解決には多大な労力を費やすことが多い。このため、性能デバグを効率的に行なうための準備は重要である。性能デバグの場合でも、測定・モデル化・制御の３つが基本であることに変わりはない。この中でも、最初に必要となる測定手段（プログラムの動きを調べる）は、後の作業の効率を左右することから、特に重要であるといえる。

このような背景から、筆者は、Linux の性能測定を目的として、イベント・トレース手法に基づいて計算機システムの性能測定・分析を行なうツール（イベント・トレーサと測定データ分析ツール）を開発した。イベント・トレーサとは、イベント（計算機内部における何らかの状態変化）を時系列として記録する測定ツールである。一般に、イベント・トレーサは、測定データが膨大な量となり、その扱いが面倒（分析結果を得るための手間

暇が大きい）という短所はあるが、システムの動作を時系列として記録しているため、測定期間中の任意の時間区間を対象にシステムの振舞いを分析でき、問題点に対して効率・効果的にアプローチできる点が長所である。

Linux 2.2 系の OS を搭載するシステムにおいて複数ファイルを並行アクセスすると性能が極端に劣化する現象が問題となっていたが、このツールを用いて分析したところ、直ちにその原因を明らかにすることができた。2.4 系の kernel では、この性能劣化現象は発生しなかったことから、当初、2.4 系の kernel で改良されたグローバルな kernel_lock が関係している可能性が高いと思われたが、分析結果は意外な原因を炙り出した。誰もが（筆者も含め）予想さえしていなかった原因を明らかにできたことから、この事例は本ツールの長所を端的に示したものと考えている。

本稿では、本イベント・トレーサの設計・実装について、他の性能測定ツールとの簡単な比較も含めて述べる。また、本トレーサが効果を発揮した事例として、Linux 2.2 系の kernel において発生した性能劣化現象（複数ファイルの並行アクセス時に発生）についての原因解明の詳細を示す。

２．イベント・トレーサの設計・実装

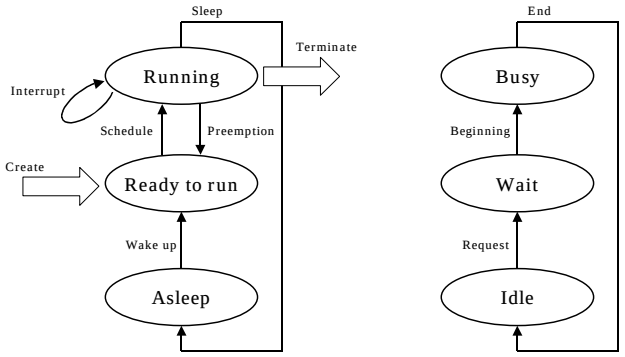
開発したトレーサは、プロセス切り換え、ディスク・アクセス、プロセス間通信といった OS に関係するソフトウエア・イベントを扱うトレーサである。ソフトウエア・イベントは、メモリ・アクセスや CPU による機械語命令実行といったハードウエアに近いレベルのイベントとは異なり、ソフトウエア技術者やシステム・エンジニア（両者を SE と総称する）がイメージするシステム動作に近いレベルのイベントである。このため、ソフトウエア・イベントのトレーサは主に SE がシステム動作を分析するために利用されるケースが多い。具体的な利用方法としては、キャパシティー・プランニングを行なうための基礎データを得る、CPU、disk、メモリ、ネットワークといった計算機資源のレベルでのボトルネックを調べる、といった性能評価を目的とした利用の他に、OS 動作（例えばプロセスのスケジューリング）に問題はないか、システムに障害が発生した際の原因究明、といった不具合の検出や分析を目的とした利用も行なわれている。

２．１　測定対象イベントの選定

イベント・トレーサを設計する際、最も重要となるのは測定対象イベントの選定である。これが、トレーサの能力（トレーサを用いた測定で分析できる性能要因の種類）を決め、更には、トレーサ自体の性能（測定オーバーヘッド、測定時間）を左右するからである。

本イベント・トレーサが検出するソフトウエア・イベントは、筆者が提案したイベント・トレースのフレームワーク[2]に基づいて選択した。このフレームワークは、計算機の構成要素である CPU、disk、メモリ、通信、プロセスをオブジェクトとみなしたとき、オブジェクト間で発生する総ての相互作用をイベントとして検出するという考え方である。例えば、プロセスと CPU の関係を図１(a)の状態遷移図、プロセスと disk の関係を図 1(b)の状

態遷移図で表したとき、これらの遷移図中に示される総ての状態遷移が CPU と disk に関する測定対象イベントとなる。



(a) CPUに関する状態遷移　　(b) Diskに関する状態遷移

図１：ソフトウエア・イベントの選定

このフレームワークは、測定対象イベントを測定データ（イベント・トレース）の分析方法（アルゴリズム）とリンクした形で規定していることから、これに基づいて測定対象イベントを選定することにより、トレース・データ分析アルゴリズムの共用、さらには、１つの測定データから様々な性能指標を得ることが可能となる。これは、測定対象イベントが、評価項目に特化した形で ad hoc 気味に、もしくは、トレーサを作成する側の視点で決められていたこれまでのイベント・トレーサにはない利点である。

２．２　実装方式

各測定対象イベントは、kernel 関数（内の特定の位置）に対応付けることができる（表１）。すなわち、これらの関数が実行されたことが測定対象イベントの発生を意味しているので、それらの関数にイベント検出のための仕掛け（プローブ）を設置すれば、イベント発生を検出できる。

表１：ソフトウエア・イベントとカーネル関数の対応

Event	Kernel function
Create	do_fork() @ kernel/fork.c
Terminate	do_exit() @ kernel/exit.c
Wake up	wake_up_process() @ kernel/sched.c
Schedule	schedule() @ kernel/sched.c (リターン点)
Preemption	schedule() @ kernel/sched.c (エントリ点)
Sleep	同上 (preemptionとはtask_structのstateで区別)
Interrupt	do_IRQ() @ arch/i386/kernel/irq.c
〃	run_bottom_halves() @ kernel/softirq.c (2.2系)
〃	do_softirq() @ kernel/softirq.c (2.4系)
Disk-request	add_request() @ drivers/block/ll_rw_block.c
Disk-beginning	scsi_do_cmd() @ drivers/scsi/scsi.c (2.2系)
〃	scsi_dispatch_cmd() @ drivers/scsi/scsi.c (2.4系)
Disk-end	scsi_done() @ drivers/scsi/scsi.c
〃	scsi_old_done() @ drivers/scsi/scsi_obsolete.c

本トレーサにおけるプローブの設置方法、すなわち、オリジナルの kernel を変更してプローブを組み込む方法として、本トレーサでは、各イベント検出点（イベントに対応するカーネル関数）には hook のみを設置し、測定操作を行なうプログラムを kernel にロード可能なモジュールとして構成する方式（図２）を採用した。これにより、hook の組み込まれた kernel は一切変更することなく、様々な方式のイベント・トレーサ（広義にはモニタリング・ツール）を実装することができる。これは、source code 中にトレーサ本体を組み込んでしまうアプローチ（例えば syscall_trace）にはないメリットである。

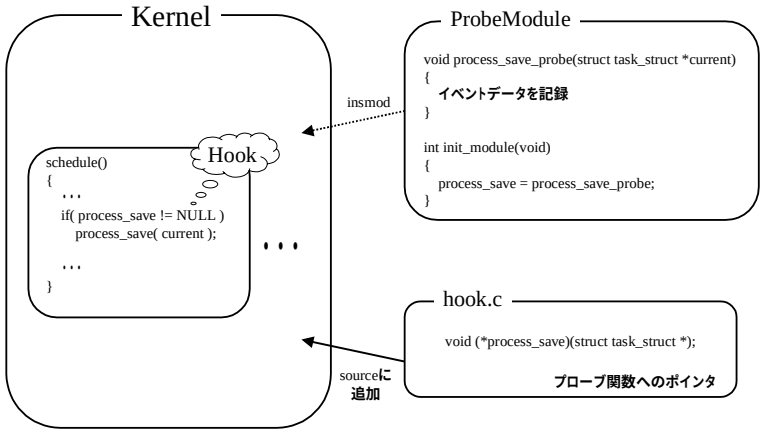


図２：ソフトウエア・プローブの実装方式

実際、この hook を利用するトレーサ２種類をインプリメントして使用している。１つは測定対象システムのメモリにトレース・データを書き込む方式のトレーサ、もう１つは測定対象システムと別のハードウエアを用いてトレース・データを記録する方式のトレーサである。後者のトレーサにおいて hook を利用するプローブ・モジュールの役割は、外付けハードウエアを接続するインタフェース・カードに対してイベント・データを書き込むことである。

２．３　測定オーバーヘッド

どのような測定であっても、測定オーバーヘッド、すなわち測定操作に起因する測定対象の乱れ、を避けることはできない。このため、測定オーバーヘッドは、測定ツールを表す特性の一つとして重要である。

イベント・トレーサにおけるオーバーヘッドは、各イベントが発生した際、被測定計算機システムの資源を用いて行なう測定操作によるものであり、その大きさは、「１イベントを処理するために使用する資源の量（使用時間）」と「イベントの発生頻度」に比例する。前者はイベント・トレーサの実装を工夫することで削減できる量であり、後者は測定対象イベントの種類によって（ほぼ自動的に）決まってしまう量である。

一般に、発生頻度の高いイベント、例えば CPU による機械語命令の実行やメモリ・アク

セスをトレース対象にすると、計算機の動作を詳細に記録することができるが、その分、オーバーヘッドが増大してしまうことになる。例えば、１命令分のイベント・データを処理するために１０命令を要するとすると、測定オーバーヘッドは１０倍（1000%）となってしまう。

本トレーサの場合、発生頻度が比較的（命令実行やメモリ・アクセスに較べると遥かに）低いソフトウエア・イベントを測定対象としているので、低オーバーヘッドでの測定が可能となっている。外付けハードウエアを併用する方式の場合、測定オーバーヘッドは 2%以下との結果[3]も得ている。オーバーヘッドが 5%以下であれば、その測定は十分に正確といわれている[4]ことを鑑みると、本トレーサのオーバーヘッドは無視できるレベルと考えて良い。

なお、本トレーサは、ハードウエア・レベルに近い（発生頻度の高い）イベントは測定しないため、ハードウエア性能の評価には適していないが、事例研究で示すような OS を含めたソフトウエア動作を評価する目的にとっては、十分に詳細なレベルのイベントを採取しているものと考えている。

２．４　他ツールとの比較

計算機の動作状況をモニタするツールには様々な方式のものがある。重要なのは、それらの特徴を把握し、目的に応じた使い分けを行なうことである。ここでは、各方式のモニタリング・ツールの特徴やそれを踏まえた適切な用途を、イベント・トレーサと対比させながら簡単に述べる。

2.4.1 サンプリング方式のツール

現在、一般的に用いられている性能モニタリング用ツールは、サンプリング方式をベースとしている、すなわち、一定間隔（10mS オーダー）で発生するクロック割り込みの度に CPU や disk といった計算機資源が使用中かどうかを調べ、それを集計していく、という基本メカニズムに基づいている。この方式によると、計算機資源の使用率、更には、特定のロック使用率や特定の関数実行時間が全体に占める割合など、使用率という尺度で表現できる性能指標を測定（推定）することができる。

この方式のツールは、インプリメントや利用方法が簡単、測定オーバーヘッドが低い、といった点が長所といえるが、使用率だけでは捉えられない現象（例えば本稿の事例で取り上げたような問題）に対しては余り効果は期待できない。測定項目が定型的な場合、例えば、常時システムの運用状況を監視するような目的で使用するのに適した方式といえる。

2.4.2 イベント・ドリブン方式のツール

イベント・ドリブン方式のツールは、測定対象システム内で発生するイベントの発生回数をカウント、もしくは、イベントからイベントまでの時間を計測するツールである。本方式によりでイベントの発生回数をカウントしておき、それをサンプリングにより一定間隔でモニタする、というような組み合わせも可能である。なお、イベント・トレーサもイベント・ドリブン方式のツールに分類されるが、本節で議論するツールには含めない。

この方式のツールは、イベントの発生回数をカウントできる、サンプリング方式よりも正確に性能指標（計算機資源の使用時間）を測定できる、という点が利点であるが、サンプル方式のツールと同様、測定項目が固定されてしまうため、様々な角度から現象を分析する目的で使用するのは不向きである。

2.4.3 イベント・トレーサ

イベント・トレーサは、システムの動作を時系列として記録するため、測定対象計算機が「なぜそのような動作をしたのか」という原因を調べる目的に有用である。特に、測定期間中の任意の時間区間を対象にシステムの振舞いを分析でき、問題点に対して効率・効果的にアプローチできる点が大きな長所といえる。その反面、イベント・トレーサは、測定データが膨大な量となり、その扱いが面倒（分析結果を得るための手間暇が大きい）という点が短所となる。

a) アドレス・トレーサ、命令トレーサ

メモリ・アクセスや CPU による機械語命令実行をトレース対象イベントとするトレーサである。一般に、このようなイベント・トレースは、cache memory や CPU のアーキテクチャを評価する目的で使用されることが多く、複数のプロセスが並行して動作する場合の性能、といった OS の観点からの評価ではない。

測定対象システムのソフトウエアを介して、ハードウエア・レベルに近く発生頻度の高いイベントをトレースしようとすると、測定オーバーヘッドが大きくなり、実働状況そのままを調べることができない。この点が問題となる場合は、トレース・データを採取する際、もしくは使用する際に工夫が必要となる。

b) ソフトウエア・イベント・トレーサ

本トレーサのように、プロセス切り換え、ディスク・アクセス、プロセス間通信といった OS に関係するソフトウエア・イベントをトレース対象イベントとするトレーサである。Linux で多用されているのは、Linux trace toolkit (LTT)[5]であろう。LTT は、本トレーサとは検出するイベントが若干異なっている（特に disk アクセス関連イベント）が、kernel 本体に hook のみを設置する方式は本トレーサと共通している。

このように種々のツールが存在している状況は、競合ではなく、各々が性能モニタリングを行なう上で重要と考えている hook 点を提案しているものと考えたい。各ツールが設置している hook を包括する形で hook 仕様を決め、（正式な kernel の source tree に組みこんだ上で）、一般公開すれば、誰でもその hook をベースとする様々な方式のモニタリング・ツールをインプリメントできるようになる。様々なツールが提供されることによって、ユーザの選択肢が広がることは、オープン・システムの持つ本質的なメリットであることから、このような方向での発展が望まれる。

３．イベント・トレーサの活用事例

ここでは、本イベント・トレーサを使うことにより、原因を速やかに解明できた性能問

題を取り上げる。まず、原因を明らかにするために行なったトレース・データの分析と結果の考察について述べる。更に、この問題の発見当初に疑いがかけられた幾つかの要因の影響についても、本イベント・トレーサによる性能分析を行ない、定量的に評価したので、その結果も示す。

３．１　性能劣化問題

Linux 2.2 系を搭載したシステムにおいて、同一 disk 上に存在する複数の大容量ファイルを並行にアクセスすると、１ファイルづつ順にアクセスした場合よりも極端に性能が悪化するという現象が見つかり、大きな問題となった。

当初、問題の原因と考えられたのは、メモリ不足による swap の頻発（thrashing)、もしくは、複数のファイルを並行にアクセスすることで disk アクセスが飛び飛びとなり、結果として disk のシーク時間が増大することである。後に 2.4.0 の kernel では性能は劣化しないということが分かり、2.4 系 kernel で大幅に改良が加えられたグローバルな kernel_lock が関係している疑いも浮上した。

３．２　分析準備

3.2.1　テスト環境

この問題を分析・解明するため、まず、現象を再現性良く発生させるテスト環境（ワークロード）を用意した。現象再現の手順を以下に示す。

1) 約 100M バイトのファイルを９個、同一ディレクトリ（disk）上に用意する。
2) １つのファイルを sequential に read（１回の read システム・コールで 64K バイト）するテスト・プログラムを用意する。このプログラムが約 100M バイトのファイル・データを read する処理を単位処理（図３参照）と考える。
3) プログラム実行前に、read 対象ファイルが属するファイル・システムを umount した後 mount することにより、そのファイル・システムに関するキャッシュ[6]を空の状態とする。
4) Step 2 で用意した９個の単位処理（プロセス）を実行させる。なお、これら９個の単位処理は、各々別のファイルにアクセスするようにしておくことで、アクセス対象データについての競合発生を避ける。

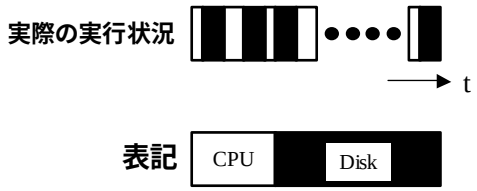


図３：単位処理
厳密には CPU と disk が交互に動作するが、
CPU 動作時間と disk 動作時間をまとめて表記する。

一旦テスト環境が整った後は、step 3 と 4 を繰り返し、同一のプログラムとファイル・セットを使用した複数のテストを行なった。（Step 1 と 2 は最初に１回行なったのみ。）

3.2.2 問題現象の再現

実験に用いたハードウエア環境を図４に示す。このマシン上で、前節に示した単位処理を１つづつ順番に実行させた場合（逐次とする）と、これらを同時に実行させた場合（並行とする）のシステム動作（図５参照）が分析対象である。図５から明らかなように、全処理を完了するまでの時間（全処理実行時間と呼ぶ）は、逐次実行の場合、各処理時間の合計、並行実行の場合、最も時間のかかったプロセスの処理時間となる。

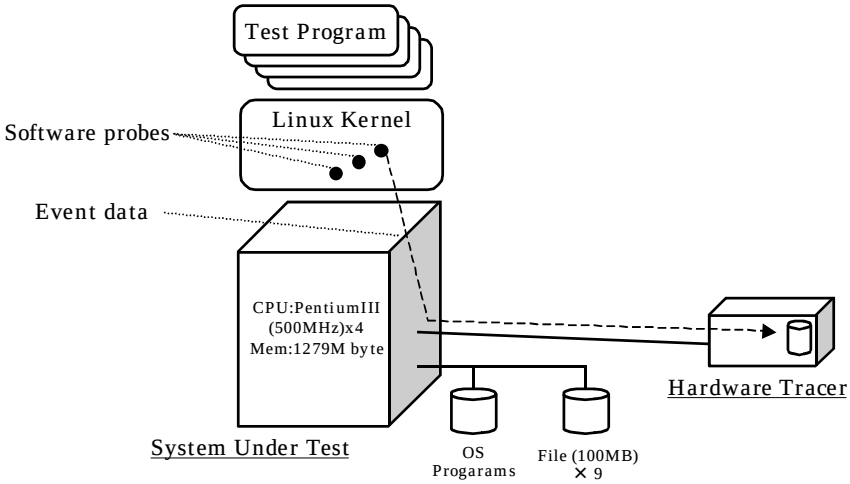


図４：測定対象システム

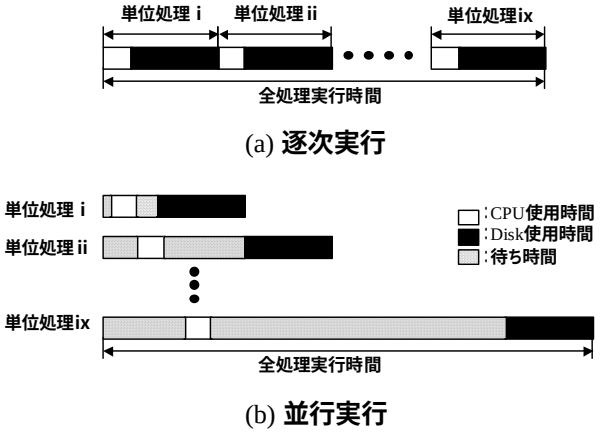


図５：動作モデル

Kernel(2.2.17)および kernel(2.4.0)上で９個の単位処理を逐次実行させたときと並行実行させたケース（計４通り）について、各処理（ｉ～ix）の実行時間と全処理実行時間を

表２に示す。

問題となった性能劣化現象を端的に示しているのが、2.2.17-並行の列、すなわち、kernel(2.2.17)で各処理を並行実行させたときの実行時間である。この全処理実行時間は、大幅に増大（逐次実行における全処理実行時間の約 10 倍）しているのに対し、kernel(2.4.0)では、並行実行と逐次実行の間に kernel(2.2.17)で見られるような大差はない。

表２：各処理の実行時間　[単位：秒]

Kernel release	2.2.17	2.2.17	2.4.0	2.4.0
実行方法	逐次	並行	逐次	並行
処理 i 実行時間	6	564	6	24
処理 ii 実行時間	6	568	6	26
処理iii実行時間	6	556	6	32
処理iv実行時間	6	568	6	48
処理 v 実行時間	6	556	6	53
処理vi実行時間	6	548	6	62
処理vii実行時間	6	557	6	73
処理viii実行時間	6	569	6	78
処理ix実行時間	6	569	6	81
全処理実行時間	55	569	55	81

３．３　イベント・トレースの採取・分析

本トレーサによりイベント・トレースを採取し、各単位処理（プロセス）の動作状況（CPU 資源の使用時間と待ち時間、disk 資源の使用時間、待ち時間、アクセス回数、アクセスしたデータの総量、１アクセス当たりの平均アクセス・サイズ）を調べた。

3.3.1　Linux 2.2.17 上の逐次実行と並行実行

Kernel(2.2.17)上で９個の単位処理（プロセス）を逐次実行させたときと並行実行させたときの結果を、それぞれ表３と表４に示す。

これらの表より、逐次実行、並行実行とも、全処理実行時間を決めているのは disk 使用時間であること、および、並行実行の性能が劣化している原因は、disk 使用時間の増大であることが分かる。CPU 使用時間が逐次実行時よりも大きくなっているのは、グローバルな kernel_lock を待つための spin lock 時間と考えられるが、性能劣化を説明できる程ではない。

Disk アクセスの状況を比較してみると、アクセスしたデータの量はほぼ同じであるのに対し、並行実行のアクセス回数が約 11 倍に増大していることが分かる。これより、並行実行の disk 使用時間の増大している原因（＝性能劣化の原因）は、disk アクセス１回当りのアクセス・サイズが小さくなることが原因であることがわかった。根本的な原因は、並行動作時に make_request 関数（drivers/block/ll_rw_block.c 内）で行なわれる disk アクセスのマージ処理（disk 上で連続するブロックへのアクセスをまとめる）がうまく機能していないことが推察される。

表３：Kernel(2.2.17)上の逐次実行　[単位：秒、K バイト]

	Elapsed	CPU 使用時間	CPU 待ち時間	Disk 使用時間	Disk 待ち時間	総アクセス回数	総アクセスデータ量	平均アクセスデータ量
処理 i	5.993	1.130	0.032	5.985	0.002	2001	104956	52.452
処理 ii	6.164	0.927	0.026	6.144	0.008	1707	104940	61.476
処理iii	6.051	0.960	0.030	6.043	0.002	2082	104940	50.403
処理iv	6.075	0.952	0.026	6.061	0.008	1692	104940	62.021
処理 v	6.069	0.982	0.030	6.055	0.008	2076	104940	50.549
処理vi	6.039	0.992	0.032	6.030	0.002	2079	104940	50.476
処理vii	6.099	0.972	0.028	6.081	0.012	1689	104940	62.131
処理viii	6.110	1.010	0.031	6.102	0.003	2076	104940	50.549
処理ix	6.075	0.992	0.029	6.059	0.010	1750	104940	59.966
合計		8.916		54.560		17152	944476	55.065

表４：Kernel(2.2.17)上の並行実行　[単位：秒、K バイト]

	Elapsed	CPU 使用時間	CPU 待ち時間	Disk 使用時間	Disk 待ち時間	総アクセス回数	総アクセスデータ量	平均アクセスデータ量
処理 i	564.469	3.318	17.092	64.555	471.311	21421	104952	4.899
処理 ii	568.033	3.328	17.522	64.021	473.815	21568	104940	4.866
処理iii	555.847	3.350	16.965	64.555	471.311	21421	104952	4.899
処理iv	568.033	3.328	17.522	64.021	473.815	21568	104940	4.866
処理 v	555.847	3.361	16.965	61.827	462.650	21640	104940	4.849
処理vi	548.275	3.291	17.087	62.056	455.089	20998	104940	4.998
処理vii	556.834	3.317	16.497	63.605	469.040	21339	104940	4.918
処理viii	568.824	3.278	18.012	63.950	476.268	21020	104940	4.992
処理ix	569.274	3.241	19.247	61.894	477.014	20137	104940	5.211
合計		29.812		570.483		191112	944484	4.942

3.3.2 Linux 2.4.0 上の動作

この問題の発見当初に疑いがかけられた幾つかの要因、すなわち、メモリ不足による swap の頻発（thrashing)、disk のシーク時間が増大、の影響についても、本イベント・トレーサによる性能分析を行なった。これらの要因の影響は、平均 disk アクセス・サイズが小さくなるという問題の起きない kernel(2.4.0)上で調べた。各々のケースについて、総実行時間および Disk アクセスに関する分析結果（９個の単位処理をまとめたもの）を表５に示す。

表５：Kernel(2.4.0)上の実行　[単位：秒、K バイト]

	総実行時間	Disk 使用時間	総アクセス回数	総アクセスデータ量	平均アクセス時間 [ミリ秒]	平均アクセスデータ量
逐次	54.504	53.695	14783	967131.136	3.687	65.422
並行	81.047	80.957	14205	967139.328	5.706	68.084
並行 (メモリ=256M	82.332	81.987	14234	967143.424	5.784	67.946
逐次 (4Kアクセス)	54.618	53.942	14777	967143.424	3.696	63.915
並行 (4Kアクセス)	80.832	80.717	14165	967139.328	5.706	66.676

a) Swap 操作の影響

使用可能なメモリのサイズを 256M バイトとした場合の結果を、表５中の「並行（メモ

リ=256M)」の行に示す。Disk アクセスの状況は「並行」の場合と殆ど同じであることから、このワークロードの場合は、swap 操作の影響はないことが分かった。

これは、本テスト・プログラムによるファイル・アクセスがシーケンシャルに read するのみ、という処理の特徴によるものと考えられる。ファイルへのアクセス・パターンがランダムな処理、または、ファイルを更新するような処理の場合は、swap 操作(ここでは disk バッファを入れ替える操作も含めた広い意味)の影響が現れることが予想される。

b) Seek 時間増大

表５の「逐次」と「並行」を比較すると、総アクセス回数と総アクセスデータ量はほぼ同じであるにも関わらず、disk 使用時間(平均アクセス時間)は、「並行」の方が長く(約 1.5 倍)なっている。この差が、disk のシーク時間増大によるものであると考えて間違いなさそうである。

c) read システム・コールのアクセス・サイズ

本事例で用いたプログラムによる disk アクセスが高速(12〜16M バイト／秒)であったのは、disk への平均アクセス・サイズが 60K バイト以上であったためである。これを大きくすることができたのは、当初、read システム・コールで指定するアクセス・サイズを 64K バイト(64K アクセスと呼ぶ)としたためと考えていたので、比較のため、read システム・コールで指定するアクセス・サイズを小さく(4K バイト)した場合について分析を行なった。結果を表５中の「逐次(4K アクセス)」と「並行(4K アクセス)」の行に示す。

当初の予想と異なり、64K アクセスと 4K アクセスに大きな差は見られなかった。これは、ファイル・システムによるプリフェッチ機能[6]が働いたものと考えられる。なお、2.2.17 についても、逐次実行の場合は、プリフェッチ機能の動作(disk へのアクセス・サイズが大きくなる)が確認できた。これより、ファイル・システムの動作は 2.2 系と 2.4 系で大きな差はない、ということも分かった。

３．４　他の事例への適用

本事例の性能劣化現象は、kernel(2.2.17)による disk へのアクセス方法が原因であったが、イベント・トレースの分析によって原因を探し出す手法は、どのような問題でも共通する基本的なものである。すなわち、分析対象となるシステム動作(単位処理)を決め、その単位処理の内容(消費するシステム資源の量、単位処理内部の実行時間など)を調べるという手法である。これにより Linux 2.2 系で HTTP のベンチマークを行なっていた際に発見した挙動不振現象も解明できた[7]ことは、この手法が広い範囲の問題に適用できることを示すものと考えている。

本事例の場合、「正常な処理」が存在していたので、これと著しく異なっている部分を探すという方法で調査できた。「正常な処理」が存在していない(データを採取できない)場合は、「正常な処理」を想定することが必要となる。このためには、例えば、ソフトウエアの設計時に性能的な要因も含めて設計し、「正常な処理」の姿を示す目安とする、といった方法が有用となろう。

４．まとめ

本稿では、Linux 用イベント・トレーサの設計、実装方式、その利用例を示した。イベント・トレーサを使うと、システムの振舞いを正確に把握できることから、特に、原因がはっきりしない予想外の動作を解明する際に有効である。このような現象が起きた場合、様々な憶測が交錯することもあり、有効な対策を執るまでに時間が掛かることが多い。有効な解決手段は、本事例研究で行なったように、問題現象を詳細に分析し、真の原因を明確にすることである。「真実はいつもひとつ」である。

ボランティア・ベースで開発が進められている Linux の場合、現象を（手軽に）正しく把握する手段を持つことは、システム開発効率化のために必須であるといえる。Open source である Linux については、様々なイベント・トレーサがインプリメントされ利用されていると考えて間違いない。今後は、個々に行なわれているこのような努力を集結し、kernel への hook 設置を実現させることが必要と考えている。

謝辞　Linux 用ソフトウエア･プローブのインプリメントに関して有益な情報ならびに議論を頂きました NEC 情報システムズ・基盤ソフトウエア事業部・高橋浩和課長（当時）と岡島順次郎主任に感謝します。また、Linux Conference 2001 への投稿に際して大変お世話になりました NEC ソリューションズ第一コンピュータソフトウエア事業部兼 OSS ソリューションセンター姉崎章博エキスパートに感謝します。

参考文献

[1] C. U. Smith, Performance Evaluation of Software Systems, Addison-Wesley, 1990.

[2] T. Horikawa, A Framework for Performance Evaluation Based on Event Tracing, IPSJ Journal, vol. 42, no. 1, pp. 68-78, Information Processing Society of Japan, 2001.

[3] T. Horikawa, TinyTOPAZ: A Hybrid Event Tracer for Unix Servers, SPECTS'99 Proceedings, pp. 203-210, The Society for Computer Simulation International, 1999.

[4] P. McKerrow, Performance Measurement of Computer Systems, Addison-Wesley, 1988.

[5] The Linux Trace Toolkit, http://www.opersys.com/LTT.

[6] 高橋、三好、図解 Linux カーネル 2.4 の設計と実装⑦―ファイルシステム（後編）、Linux Japan, vol. 4, no. 5, pp. 139-165, 五橋研究所, 2001.

[7] T. Horikawa, A Performance Measurement and Analysis Method based on Event Tracing, to be appeared in CMG 2001 proceedings.