情報処理学会論文誌　コンピューティングシステム　Vol. 1　No. 1　133–143 (June 2008)

プロセストレース機構の多重化法

川崎仁嗣[†1]　阿部洋丈[†2]　加藤和彦[†1]

プロセストレース機能を利用したソフトウェアとしては，デバッガや仮想化ソフトウェア，セキュリティソフトウェアなどがある．これらのソフトウェアを組み合わせて動作させることは，現在のプロセストレース機能の仕様上，ソフトウェアの実装を改変する必要がある．本研究ではプロセストレース機能の動作をエミュレーションすることで，これらのソフトウェアを改変することなく組み合わせて動作可能にする手法を提案する．この手法の特徴は，プロセストレース機能を利用してプロセストレース機能の動作を拡張していることである．そのため，ユーザモードレベルで動作可能であり，カーネルを改変する手法よりも移植性が高い．

A Nesting Technique of a Process Trace Mechanism

Satoshi Kawasaki,[†1] Hirotake Abe[†2]
and Kazuhiko Kato[†1]

Process trace mechanisms in existing operating systems do not allow multiple tracing processes at a time on one process although such mechanisms are used in various systems including debuggers, security systems, and resource virtualization. The method proposed in this paper emulates a process trace facility and enables the composite use of these softwares without modification. One of the main features of this method is to extend the process trace facility itself. This method is operable in user-mode level and is easier to port than modifying the kernel.

†1 筑波大学システム情報工学研究科
Graduate School of Systems and Information Engineering, University of Tsukuba
†2 豊橋技術科学大学
Toyohashi University of Technology

1. はじめに

今日，一般的に利用されている UNIX 系 OS や Windows OS にはプロセストレース機能を提供するものがある．プロセストレース機能とは，プロセスの内部状態を取得したり，任意の時点で停止させたりする機能である．また，この機能には実行状態取得だけでなく，メモリ領域や CPU のレジスタを書き換える機能もあわせ持つ場合が多い．このような機能が OS に実装された元々の理由はデバッグ機能を実現するためである．しかし，プロセストレース機能は柔軟性が高く，デバッガの実装だけではなくそれ以外の用途にも利用されており，例をあげると，セキュリティの向上[1] やシステムリソースの仮想化[2)–4)] などの分野で用いられている．

プロセストレース機能はデバッガ以外の用途でも使われるようになったが，これは本来の用途では想定されていなかった使い方である．同機能は元々，デバッガを実装するために設計された機能であるため，複数のプロセスが 1 つのプロセスをトレースすることを考慮していなかった．このため，1 つのプロセスをトレースできるプロセスは 1 つに限られている．つまり，プロセストレース機能を利用したソフトウェアを複数併用することは難しい．しかし，現在ではプロセストレース機能がデバッガだけではなく，上で述べたような用途にも利用されている．仮想化ソフトウェアやセキュリティソフトウェアを組み合わせて動作させたい場合も出てくると考えられる．この組合せに限らず，プロセストレース機能を利用しているソフトウェアどうしを併用したいという状況も十分に考えられる．

トレースされているプロセスがシステムコールを呼び出そうとする際にトレースを行っているプロセス（トレーサ）へ通知が送られる．しかし，この通知が 1 つのプロセスにしか配送されないため，トレースするプロセスが 1 つでないとうまく動作しない．本研究では，プロセストレース機構を多重化可能なように拡張することでプロセストレース機能を持ったソフトウェアを併用できるようにする．これには，トレースを専門に行うトレースマネージャを用意し，それぞれのトレーサはトレースマネージャを介してトレースを行う．トレースマネージャは受け取った通知を，トレーサに分配する．

本稿の構成として，2 章でプロセストレース機能を利用したソフトウェアを組み合わせて利用する例をあげる．その後，3 章で提案手法を示し，4 章で実装の詳細を述べる．5 章では多重化により費されるコストを測定し，考察する．

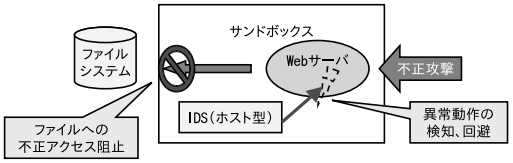

図 1　IDS とサンドボックスの組合せ
Fig. 1　Combination of IDS and SandBox.

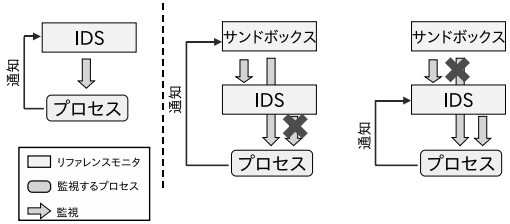

図 2　リファレンスモニタのネスト化
Fig. 2　Nesting reference monitors.

2. 多重化の例

プロセストレース機能を用いたセキュリティソフトウェアには，侵入検知システム（IDS）などのリファレンスモニタがある．仮想化ソフトウェアとしては，ファイルシステム空間を仮想化するサンドボックスや OS 全体を仮想化する User Mode Linux（UML）[2] などが有名である．

プロセストレース機能を利用しているソフトウェアは多数あるが，本研究ではそれらのシステムを組み合わせて使うことを可能にする．ここでは典型的な組合せの例を示し，それにより得られる利点を述べる．

2.1　セキュリティソフトウェアどうしの併用

サーバのセキュリティを向上させるための手法の 1 つに IDS の利用があげられる．しかし IDS はその特性上，完全な防御は難しい．学習型のモデルの場合，学習不足により本来は異常とすべき状態を正常と判断してしまうことがある．また，静的解析のモデルであっても IDS の持つ異常検知の性質を逆手に利用して，正常動作に見せかけながら攻撃を行うことが可能であり，mimicry attack [5] と呼ばれる手法が存在する．それゆえ，IDS を単体で用いていても完全に侵入を防ぐことはできない．また，IDS による保護が不完全である点だけでなく，IDS に脆弱性が存在する場合が考えられる．IDS によりソフトウェアが保護されていたとしても，IDS 自体に脆弱性があれば IDS を乗っ取られてしまう．そこで，IDS をサンドボックス [3),6)] 内で動作させることで，任意のファイルにアクセスしようと試みてもそのアプリケーションが元々アクセスできる資源以外にはアクセスできないようにする（**図 1**）．

サンドボックスにより隔離している環境の中でソフトウェアを動作させ，そのソフトウェアを IDS で監視する場合，サンドボックスと IDS はネスト関係になる．つまり，サンドボックスは IDS のプロセスと監視対象プロセスを監視し，IDS は監視対象プロセスを監視する（**図 2**）．すでにサンドボックスで監視されているプロセスを IDS で監視しようとした場合，すでにトレースされているプロセスへのトレース要求は失敗するため，IDS がプロセスを監視することはできない．同様に，IDS で監視されているプロセスをサンドボックスで監視しようとしても失敗する．

本研究での提案手法を適用することで，IDS やサンドボックスなどのリファレンスモニタを複数組み合わせて動作させることができる．これによりそれぞれのリファレンスモニタの持つ欠点を補うことが可能である．

2.2　仮想化ソフトウェアとデバッガの併用

プロセストレース機能では，プロセスから OS へのシステムコールを検知し，実際のシステムコールが実行される前後で任意の処理を行うことができる．UML ではこれを利用し，UML 内プロセスに対して仮想化された資源を提供する．UML では，UML 内で動作するプロセスが UML カーネルによってトレースされている．これはプロセスが呼び出すシステムコールを検知し，UML カーネルがシステムコールを処理するために行われている．また，UML カーネルのカーネルスレッドもトレースされている．したがって，ホスト OS 上のデバッガから UML 内のプロセスや UML カーネルのデバッグをすることは本来できない．しかし，UML カーネルでは ptrace proxy と呼ばれる機構が実装されており，デバッガを動作させることができるようになっている．これについては 6 章で詳しく述べる．

UFO [4] と呼ばれるユーザレベル仮想ファイルシステムでは，FTP や HTTP 上のファイルをローカルファイルシステム空間にマッピングする．UFO では，ローカルファイルシステムへのアクセスをリモートファイルへのアクセスに変換する．UFO システムは，UFO を利用するプロセスをトレースすることでファイルシステムへのアクセスを検知している．このため，UFO を利用しているプロセスに対してデバッガからトレースを行うことはできない．

本研究での提案手法を適用することで，UFO などのプロセストレース機能による資源仮想化ソフトウェアを利用したプロセスをデバッグすることが可能となる．

3. 提案手法

この章では，プロセストレース機能とそれを利用するプロセスとの間に通知の分配機構を導入し，トレーサプログラムを多重に動作可能にする手法を述べる．

はじめに，プロセストレース機能を利用した動作の流れについて述べる．**図 3** で，サービス呼び出しが行われる（①）と OS のプロセストレース機能によりトレーサプログラムへ通知が行われる（②）．このとき，監視対象プログラムは動作を一時停止した状態にある．トレーサプログラムは通知を受け取ると，監視対象プログラムの呼び出そうとしていたシステムコールの番号や引数を取得したり，メモリ領域のデータ読み出しや変更をしたりすることができる．実行を再開する場合は OS へ実行を継続するよう指示し（③），OS は実行結果を返す（④）．

本研究で対象とするプロセストレース機構の機能を以下に示す．

(1)　プロセスの任意点での実行中断
(2)　実行中断時の通知機能
(3)　プロセスのメモリ領域の読み書き
(4)　プロセスのレジスタ情報の読み書き
(5)　プロセスの実行再開

(1) の任意点とは，プロセスのシステムコール呼び出し時，他プロセスからのシグナル受け取り時などである．プロセストレース機構がここで示した機能を提供していれば，本提案手法を適応可能と考えられる．UNIX 系 OS の多くは，ここに示した機能を提供しているため，以降では UNIX 系 OS を対象として提案手法を述べる．

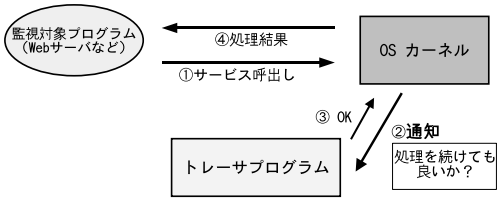

図 3　プロセストレース機構の動作
Fig. 3　Mechanism of the process trace facility.

プロセストレース機能を用いたトレーサプログラムの場合，2 章で述べたように，すでにトレースされているプロセスに対して多重にトレースを行うことはできない．プロセスをトレース中にシステムコールの呼び出しが行われた場合，カーネルはそのプロセスの親プロセスに対してシグナルを送り通知するが，あるプロセスに対する親プロセスは 1 つである．したがって，トレーサプログラムを複数動作させた場合，実際に通知を受け取ることのできるプロセスは 1 つだけであり，通知を受けることのできなかったプロセスはトレース動作を行うことができない．たとえば，図 2 ではサンドボックスがトレースを行っているプロセスに対して，IDS からトレースを行うことはできない．同様に，IDS がトレースを行っているプロセスをサンドボックスがトレースすることはできない．ただし，1 つのプロセスが複数の子プロセスに対してプロセストレースを行うことは可能である．この場合，複数の子プロセスからの通知を 1 つの親プロセスが処理する形となる．

3.1 概　　要

複数プロセスによるトレース動作を実現するために本研究では，プロセストレースシステムコールを直接利用するのではなく，プロセスのトレースを専門に行うトレースマネージャ（**図 4**）を用意し，実際のプロセストレースはトレースマネージャが一括して行う方法を提案する．トレースマネージャの役割は大きく分けて 3 つある．

(1)　通知の受け取り
(2)　通知の分配
(3)　トレースインタフェースの互換性維持

図 4 に動作の流れを示す．サービス呼び出しが行われると（①），OS のプロセストレース機構により通知がトレースマネージャに送られる（②）．トレースマネージャは通知を送っ

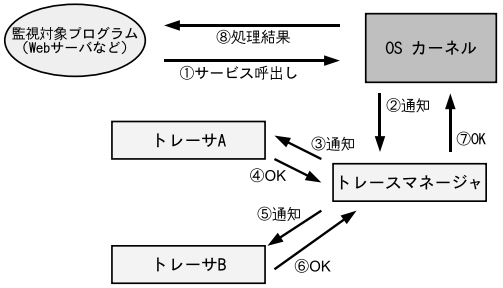

図 4　トレースマネージャ
Fig. 4　Trace Manager.

てきたプロセスを監視しているプロセスがあれば，通知を転送する．トレーサ A は通知を受け（③），実行を許可する場合は実行を継続するよう指示する（④）．ほかにも監視を行っているプロセスがいる場合，通知を転送する．トレーサ B は通知を受け（⑤），実行を許可する場合は実行を継続するよう指示する（⑥）．他に監視を行っているプロセスがなければ，OS に実行を継続するよう指示する（⑦）．OS は処理結果を返す（⑧）．以下で，これらの動作について詳しく述べる．

通知はサービス呼び出しをきっかけにして OS が生成するものであるが，以降では説明の簡略化のため，サービス呼び出しを行ったプロセスが通知を生成したように表記する．

3.2　通知の受け取り

トレースマネージャが通知を分配するためには，まず，通知がトレースマネージャに送られてくるようにする．2 章で述べたとおり，通知は（トレースを行っている）親プロセスに対して送られる．したがって，トレース対象プロセスをトレースマネージャの子プロセスとする．これはトレース対象プロセスをトレースマネージャから実行すれば子プロセスとなるので問題ない．しかし，トレース対象プロセスが新たにプロセスを生成する場合，そのプロセスの親プロセスはトレースマネージャではなく，トレース対象プロセスとなってしまう．また，生成されたプロセスはトレース状態ではなくなってしまうため，プロセストレース機構によって追跡されなくなってしまう．新たに生成されたプロセスから通知を直接受け取るためには，トレースマネージャが親プロセスとなり，プロセスをトレース状態にする．

サンドボックス内でホスト型 IDS を動作させた場合の例を説明する．リファレンスモニタ上で動くプロセスは**図 5** の左に示すようなプロセスの親子関係であるかのように動作する．見掛け上，プロセス A は IDS とサンドボックスに監視されているのでそれぞれに通知を送り，IDS はサンドボックスに監視されているのでサンドボックスに通知を送るように見える．しかし，実際には図 5 の右に示すように，すべてのプロセスはトレースマネージャの直接の子プロセスとなっている．プロセス A からの通知はトレースマネージャが受け取り，プロセス A を監視している IDS とサンドボックスに通知が伝えられる．IDS からの通知もトレースマネージャが受け取り，サンドボックスに伝えられる．

すべてのプロセスをトレースマネージャの子プロセスにするため，リファレンスモニタやトレース対象プロセスが子プロセスを生成すると，その処理をフックして，トレース状態であることを示すフラグや親プロセスの情報を書き換える．この操作により，プロセスからの通知はすべてトレースマネージャに送られる．なお，ここで書き換える親プロセスの情報とは通知の送付先を示す情報であり，カーネル内では本来の親プロセスの情報も別途保持されている．

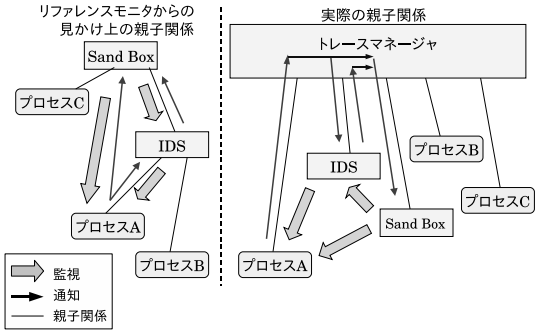

図 5　プロセスの親子関係
Fig. 5　Processes parental relation.

3.3　通知の分配

トレースマネージャは OS カーネルから通知を受け取ると，その通知を本来受け取る予定であったプロセスに対して送り出す．通知を受け取るためにトレーサプロセス（トレースを行っているプロセス）は，待ち受けシステムコールを発行する．このシステムコールを呼び出すと，トレースマネージャから通知が到着するまでプロセスはブロックされる．また，このシステムコールを呼び出す際に，どのプロセスからの通知を待ち受けるか指定することができる．したがって，トレースマネージャは通知を転送する前に，トレーサプロセスをブロック状態から復帰させる．また，通知は復帰するときの返り値として渡されるため，返り値を書き換えて子プロセスから通知を受け取ったように動作させる．複数のトレーサプロセスに対して同時に通知を送ってしまうと，トレーサプロセスは本来 1 つしかない前提で動作しているため，正常な動作ができなくなってしまう．整合性を保つために，通知は同時に 1 つのトレーサプロセスにしか送られない．図 4 に示すように，トレーサプロセスが動作を終えると，他のトレーサプロセスの 1 つに通知が送られる．トレーサプロセスすべての動作が終わるまで，この処理が繰り返される．

これらの動作を行うために，トレースマネージャはプロセスのトレース関係を管理している．これは，プロセストレースシステムコールが呼び出された際に，どのプロセスをトレースしようとしているのかを記録することで実現できる．

3.4　プロセストレースインタフェースの互換性維持

既存のリファレンスモニタを修正しなくても動作可能とするために，トレースマネージャは大きく分けて 4 つのシステムコールの動作をエミュレーションする．1 つ目は，3.2 節で述べたように，プロセス生成系のシステムコールをフックする．これは，トレースマネージャが通知を受け取り可能とするためである．

2 つ目は，3.3 節で述べたように，通知を待ち受けするシステムコールにフックをかける．これは，本来の通知受け取りではトレース対象プロセスから通知を受け取ったように見えるのと同様に，トレースマネージャ経由でもトレース対象プロセスから通知を受け取ったように見せるためである．

3 つ目は，通知ハンドラ操作系のシステムコールのフックである．これは，トレースマネージャが通知を処理するため，その通知に対するハンドラをカーネルに設定しても，処理されないためである．トレースマネージャはカーネルに代わってハンドラの処理を行う．

4 つ目はプロセストレースを操作するためのシステムコールのフックである．これは，実際の親プロセスがトレースマネージャであるため他のプロセスはプロセストレースの操作を行えないためである．したがって，トレースマネージャが状態を取得し，それを受け渡す必要がある．トレーサプロセスがレジスタの情報を取得する操作を行おうとすると，トレースマネージャが代わりにレジスタの情報を取得し，それをトレーサプロセスに渡す．ほかにも，1 つのトレース対象プロセスに対して複数のプロセスがトレースを行うことで，本来ならば 1 回しか起きない動作を複数回行おうとするため，不整合が生じてしまう場合がある．たとえば，子プロセスの実行を再開させる指示はプロセストレースシステムコールで行うが，トレーサプロセスのすべてが動作を終えてから子プロセスを再開させたいので，実行の再開はすべてのトレーサプロセスが動作を終えてからトレースマネージャが行う．つまり，複数のトレーサプロセスが動作していても，3.3 節で述べたように同時に動作するトレーサプロセスは 1 つである．そのトレーサプロセスが子プロセスを実行再開するよう指示を出した場合でも，図 4 に示すようにまだ通知を受けていないトレーサプロセスがある場合には子プロセスの再開処理は行わない．すべてのトレーサプロセスが通知を受けて動作し終わっている状態であれば，再開処理を実行する．

この動作を実現するためにプロセストレースシステムコールをフックし，子プロセスに対する操作を代替したり，整合性を保つような処理を行ったりする．

これらの動作により，トレースマネージャはプロセストレースシステムコールの動作をエミュレーションする．したがって，トレーサプロセスは間にトレースマネージャが介在していることを意識しなくてもよい．

3.5　議　　論

本提案方式ではプロセストレース機構を利用して，プロセストレース機構の動作を拡張している．プロセストレース機構を利用することで，トレースマネージャをユーザモードかつユーザ権限で動作させることができる．またカーネルのバージョンなどによってプロセストレース機構の動作が大幅に変わってしまうこともないため，トレースマネージャの移植性は高い．しかしながら，ユーザモードで動作するためカーネルモード内での拡張よりもオーバヘッドが大きくなってしまう．さらに，プロセストレース機構自体も動作が遅く，システムコールを頻繁に呼び出す処理では動作速度の低下が問題になる場合がある．また，多重化の段数が深くなるほど，オーバヘッドは指数関数的に増えてしまう．多重化の段数を n としたとき，コンテキストスイッチの発生回数は，$2^n + 2$ となる．しかし，本提案手法では多重化の段数を増やさなくても複数のトレーサを動作させることができる．その場合，トレーサどうしでの監視はされないが，その分オーバヘッドを低減することが可能である．5 章で実験によりオーバヘッドを測定する．

4. 実　　装

本章では，3 章で述べた提案方式を UNIX 系の OS 上で実装する方法について述べる．UNIX 系の OS で提供されるプロセストレース機能は，ptrace や procfs などのインタフェースを介して利用可能である．OS で提供されるインタフェースの種類を**表 1** に示す．本章では ptrace を用いた実装について述べる．procfs を利用した場合でも，ほぼ同様の手法が利用できる．

ptrace システムコールを利用したプロセス監視において，UNIX 系 OS では通知方法としてシグナルが利用されている．通知の分配などは，シグナルの分配として実装する．

トレースマネージャはリファレンスモニタが呼び出す ptrace システムコールをフックし，

表 1　プロセストレース機能のインタフェース
Table 1　Interface of process trace facility.

Linux	ptrace
BSD 系 UNIX	ptrace および procfs
Solaris	ptrace ⋆1 および procfs

⋆1 Solaris における ptrace は procfs を利用してライブラリ上で実装されている．

表 2　エミュレーションの必要なシステムコール
Table 2　Systemcalls required emulation.

プロセス生成	fork
	vfork
	clone
シグナル待ち受け	wait
	waitpid
	waitid
	wait3
	wait4
シグナルハンドラ操作	sigaction
	signal
プロセストレース	ptrace

監視動作をエミュレーションする．しかし，3 章で説明したとおり，ptrace システムコール以外にもいくつかのシステムコールをフックし，エミュレーション処理を行う．fork システムコールに代表されるプロセス生成の操作や wait システムコールに代表されるシグナル待ち受けの操作，sigaction システムコールに代表されるシグナルハンドラの操作，そして ptrace システムコールなどのプロセストレースシステムコールである．3 章で述べたシステムコールに加え，シグナルハンドラ操作についてもエミュレーション処理を行う必要がある．なぜならばシグナル処理はトレースマネージャが行うため，そのシグナルに対してどのような動作をするかをトレースマネージャに教える必要があるからである．Linux の場合，エミュレーションの必要なシステムコールは**表 2** に示すとおりである．

以下，これらのエミュレーション手法について説明する．

4.1　プロセス生成

プロセス生成の操作（fork など）が行われる場合，新たに生成されたプロセスがトレースマネージャの子プロセスでないとトレースマネージャへシグナルが送られてこない．さらに，新たに生成されたプロセスはトレース状態ではないので，生成されたプロセスをトレース状態にする必要がある．これは ptrace システムコールを利用してアタッチ（動作中のプロセスをトレース状態に変更し，通知を発生するよう設定すること）することで実現できるが，カーネルのスケジューリングのタイミングによって生成されたプロセスがアタッチされる前に実行されてしまう可能性がある．そこで，プロセスの生成を行うシステムコールが呼び出されたとき（**図 6**.i）は，それらのシステムコールを clone システムコールに書き換え，トレース状態をコピーするように指定するフラグを追加する（図 6.ii）．その後，プロセス

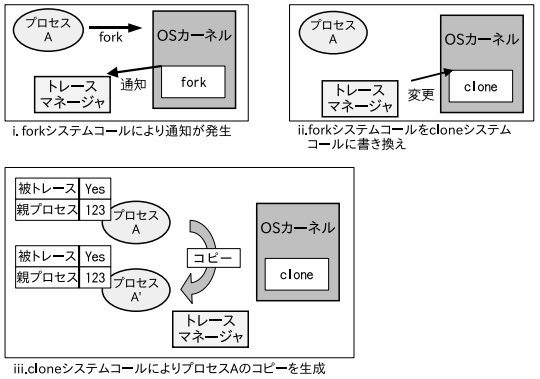

図 6　プロセス生成の処理
Fig. 6　Emulation of process creation.

は clone システムコールを実行し，新たなプロセスが生成される．このとき，元のプロセスのトレース状態は生成されたプロセスにコピーされるため，トレースマネージャが親プロセスとなりトレースを行える（図 6.iii）．

4.2　シグナル待ち受け

リファレンスモニタはシグナル待ち受けの操作（wait など）を行い，シグナルが送られてくるまで待機する．シグナルが送られてくると，待機状態から復帰しシグナルを受け取る．また，wait はシグナルの待ち受けだけでなく，子プロセスの状態変化を検知するためにも利用される．

トレースマネージャを用いた環境では，親プロセスがトレースマネージャなのでその中で動作しているリファレンスモニタや監視対象プロセスは wait システムコールを呼び出しても，待機状態から復帰できない．そこで，トレースマネージャ側で wait システムコールを検知したとき（**図 7**.i）は，どのプロセスの状態変化を待っているか調べておき（図 7.ii），条件にあったプロセスで状態変化があった場合，待機しているプロセスの実行を再開させる（図 7.iii）．待機プロセスはブロック状態を解除され，トレースマネージャが受け取ったシグナルを受け取ることができる．

4.3　シグナルハンドラ操作

シグナルハンドラの操作（sigaction など）が行われると，そのプロセスがシグナルを受け取ったときの動作を変更することができる．しかし，トレースマネージャが動作している

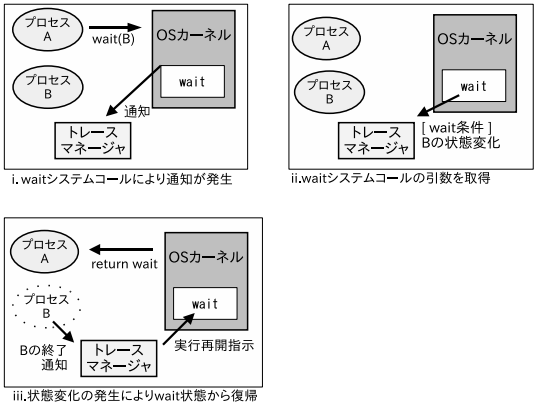

図 7　プロセス間同期の処理
Fig. 7　Emulation of process synchronous.

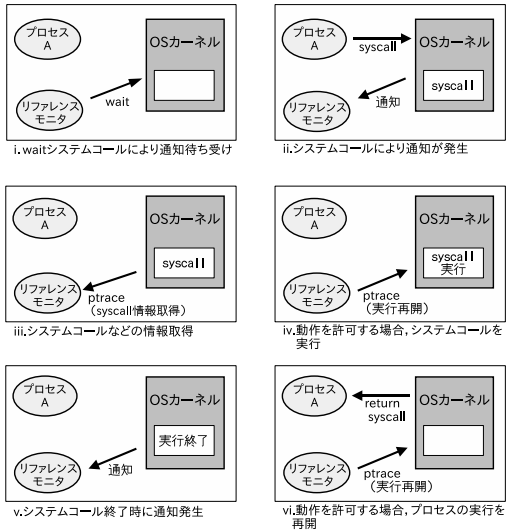

図 8　通常のプロセス監視の流れ
Fig. 8　Original process trace.

場合，監視対象プロセスはシグナルを受け取るとそのことがトレースマネージャに通知されて監視対象プロセスは停止する．受け取ったシグナルの処理はトレースマネージャで行われるため，シグナルハンドラの操作を検知した場合，トレースマネージャはシグナルを受け取ったときにどのように動作するかを記録しておく．監視対象プロセスがシグナルを受け取り停止した場合，トレースマネージャはそのシグナルに対応する動作が設定されているか確認し，設定されていればプロセスの動作を再開させシグナルを渡す．設定されていなければ，既定の動作（プロセスの停止など）を行う．

4.4　プロセス監視

ptrace システムコールによるプロセス監視では，親プロセスが wait システムコールにより子プロセスからのシグナルを待ち（**図 8**.i），子プロセスはシステムコールの直前で停止し，シグナルが親プロセスに送られる（図 8.ii）．このシグナルにより親プロセスは wait から復帰し，子プロセスの状態を調べる（図 8.iii）．このとき，子プロセスのレジスタやメモリの状態を取得し，システムコールの種類や引数などを調べることができる．たとえば，サンドボックスの場合は open システムコールに代表されるファイル操作関連のシステムコールの引数を監視する．そして，アクセスポリシに基づきシステムコールの実行を継続してよいか判断する．実行を継続するよう指示を出すと，OS カーネルがシステムコールを実行する（図 8.iv）．システムコールの処理が終了すると停止し，シグナルが親プロセスに送られる（図 8.v）．このとき，システムコールの返り値を取得，書き換えを行うことができる．実行を継続するよう指示を出し，子プロセスの動作を再開させる（図 8.vi）．

トレースマネージャによる管理下でリファレンスモニタやプロセスを動作させた場合について説明する．

モニタプロセスは wait システムコールを呼び出し，シグナル待機状態になる（**図 9**.i）．プロセス A がシステムコールを発行すると，システムコール直前で停止し，トレースマネージャにシグナルが送られる（図 9.ii）．トレースマネージャはそのプロセスを監視しているモニタプロセスに対して，シグナルを送り wait システムコールによるブロック状態から復帰させる．すると親プロセスは wait システムコールから戻り，その時点で停止する．ここで wait の返り値を，監視しているプロセスのプロセス ID に書き換える．これによりモニタプロセスは監視していたプロセスが停止してシグナルを送ってきたように見える（図 9.iii）．モニタプロセスはプロセスの状態を取得するために ptrace システムコールを呼び出す．モニタプロセスもトレースされているので停止し，トレースマネージャにシグナルが送られる（図 9.iv）．モニタプロセスからはプロセス A をトレースしているように見えるが，実際に

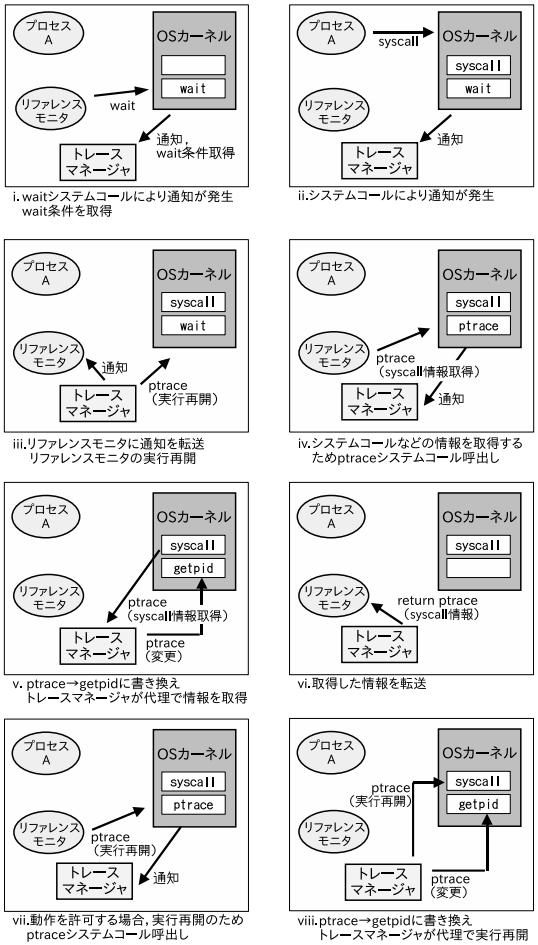

図 9　プロセス監視のエミュレーション
Fig. 9　Emulation of process trace.

はトレースマネージャがトレースを行っている．したがって，プロセスの状態を取得する動作もトレースマネージャでなければ行えない．ここでのプロセス情報取得はトレースマネージャが代理で行う．また，この ptrace システムコールはエミュレーションするものであり，実際に実行させてしまうと不都合が生じる．そこで，トレースマネージャは ptrace システムコールを（後続の処理に）副作用のないシステムコール呼び出し（getpid など）に書き換える（図 9.v）．getpid システムコールを実行させ，返り値は ptrace の実行結果に書き換える（図 9.vi）．モニタプロセスは取得したプロセス状態をもとに実行を継続してもよいかどうかを判断する．実行を継続する場合は ptrace システムコールを呼び出す．モニタプロセスもトレースされているので停止し，トレースマネージャにシグナルが送られる（図 9.vii）．トレースマネージャはプロセス A の実行を継続するよう OS に指示する．先ほどと同様に，トレースマネージャは ptrace システムコールを副作用のないシステムコール呼び出しに書き換え，実行させ，返り値は ptrace の実行結果に書き換える（図 9.viii）．

ここまでで，プロセス A のシステムコールが OS カーネルで実行される部分まで処理が進んだ．これは図 8.iv までに相当する．システムコール終了時にも同様の手順で監視処理が行われる．

4.5　システムの利用方法

本章で実装したシステムの利用方法を説明する．基本的には，以下に示すようにトレースマネージャ（trace_mng）起動時にトレーサ（$tracer_A$，$tracer_B$），監視対象プロセス（target_process）の場所を指定し，それぞれの監視対象プロセスごとに個別にトレースマネージャを起動する．

$ $trace\_mng\ tracer_A\ tracer_B\ target\_process$

トレーサを指定するときに，先に書いたもの（$tracer_A$）は後に書いたもの（$tracer_B$）を監視する．トレースマネージャは指定されたプロセスを起動し，監視を開始する．すでに動作しているプロセスやトレーサに対して，後からトレースマネージャを組み込むときにも同じような方法でトレースマネージャを起動する．その際は，プロセスの場所ではなくプロセス ID を指定する．

すでに動作しているプロセスをトレースマネージャの管理下にすることができるので，1 つのプロセスを 1 つのトレーサが監視していて，後にトレーサマネージャを組み込んでトレーサを 2 つにすることも可能である．これにより 1 つしかトレーサが動作していないときに無駄なオーバヘッドをかけずに済む．

5. 実　　験

提案手法による実装を行い，多重化を行うためにかかるコストを調査した．実装の環境は Linux 2.6.20（i386）上であり，調査を行った環境は CPU が Core 2 Duo 2.0 GHz，メモ

表 3　実行時間（IDS ＋ sandbox）
Table 3　Processing time (IDS ＋ sandbox).

監視対象プログラム	(i) なし	(ii)IDS	(iii) トレースマネージャ ＋ IDS	(iv)sandbox	(v) トレースマネージャ ＋ sandbox	(vi) トレースマネージャ ＋ IDS ＋ sandbox
make	12.06	12.85	15.26	13.16	15.30	15.83 (1.88)
platex	0.13	0.17	0.61	0.14	0.60	0.81 (0.63)
dvipdfmx	1.99	2.43	9.85	2.49	10.32	12.89 (9.96)
lhttpd	0.59	0.83	1.17	0.77	1.20	1.40 (0.39)

単位：秒
(vi) の括弧内の数値はトレースマネージャ自身の処理時間

リが 2 GB である．

監視対象のプログラムとして，以下の 4 つを用い，それぞれの処理に要する時間を測定した．

（1） make
　　トレースマネージャのソースコードをビルドするのに要する時間
（2） platex
　　1,000 行ほどの TEX ファイルのコンパイルに要する時間
（3） div2pdfmx
　　60 kB ほどの dvi ファイルを PDF ファイルに変換するのに要する時間
（4） light HTTPD
　　80 kB ほどのページを wget で 100 回取得するのに要する時間（light HTTPD と wget は同一計算機上で実行）

監視プログラムとして，IDS と sandbox を利用した．ここで用いた IDS は，システムコールの発行順序を bi-gram でモデル化し，これをもとにプログラムの動作を監視する．また，sandbox は open システムコールを監視し，ポリシに反するファイルへのアクセスは拒否する．

条件を以下の 6 つにして実験を行った．

（i） 他のプロセスから監視されていない状態で実行した場合
（ii） IDS により監視されている場合
（iii） トレースマネージャを利用して IDS により監視されている場合
（iv） sandbox により監視されている場合
（v） トレースマネージャを利用して sandbox により監視されている場合

表 4　システムコールの割合
Table 4　The percentage of system calls.

プログラム	システムコール数	全体に占めるシステムコール処理時間
make	99741	7.15%
platex	1328	10.39%
dvipdfmx	37918	14.22%
lhttpd	15468	-

（vi） トレースマネージャを利用して 2 つのトレーサ（IDS ＋ sandbox）を同時に適用した場合

(vi) の 2 つのトレーサのうち，IDS が監視対象のプログラムをトレースし，sandbox は IDS と監視対象のプログラムをトレースする．

実際に測定した結果を**表 3** に示す．

結果の (vi) の括弧内の数値はトレースマネージャ自身の処理時間である．オーバヘッドの割合が低いのは make や lhttpd であり，オーバヘッドの割合が高いのは platex や dvipdfmx である．これはアプリケーションによってシステムコールを呼び出す頻度が異なることが原因の 1 つである．**表 4** に示すように，make に比べ platex や dvipdfmx は全体の処理時間のうち，システムコール処理にかかる時間が多い．また，1 つのシステムコールあたりの処理時間は，平均するとほぼ同じであることから，システムコール処理時間とシステムコールの呼び出し回数は比例関係であると考えられる．したがって，システムコール呼び出し回数は $make < platex < dvipdfmx$ であり，システムコール呼び出し回数の多い platex や dvipdfmx はオーバヘッドが大きい．本提案手法はシステムコールを大量に呼び出すプログラムに不向きであると考えられる．

表 5　コンテキストスイッチ回数
Table 5　Count of context switches.

監視対象 プログラム	(iii) トレースマネージャ ＋ IDS	(v) トレースマネージャ ＋ sandobox	(vi) トレースマネージャ ＋ IDS ＋ sandbox
make	414498 (207248)	419538 (209768)	442578 (221288)
platex	47832 (23915)	43600 (21799)	67690 (33844)

括弧内の数値はトレースマネージャが通知を受け取った回数

オーバヘッドの原因として，ほかにはコンテキストスイッチが考えられる．システムコールの呼び出しが起こるたびにトレースマネージャに通知が送られ，さらにトレースマネージャは通知をトレーサに転送する．その際にコンテキストスイッチが多発してしまうため，オーバヘッドが大きくなってしまう．make と platex について，コンテキストスイッチ数とトレースマネージャの通知受け取り回数を**表 5** に示す．コンテキストスイッチ回数の増加に合わせて，表 3 に示す実行時間が増加しているのが分かる．本章での評価では，トレーサを 2 段に多重化したが，本提案手法ではそれ以上の多重化も実現できる．しかし，3.5 節で述べたように，コンテキストスイッチの回数は多重化の段数を n としたとき，$2^n + 2$ となる．したがって，本章の評価結果からすると現実的に利用可能な多重化は 2 段までであると考えられる．一方，多重化は 2 段のまま，トレーサプロセスを 3 つ以上動かすことも考えられる．この場合，トレーサプロセス数が増えるとコンテキストスイッチも増加してしまうが，多重化の段数を増やす場合に比べればコンテキストスイッチの回数は抑えられる．

本提案手法では実現方式が複雑なため，オーバヘッドを小さく抑えるのは難しい．しかしながら本提案手法を利用することで，プロセストレース機能を利用したソフトウェアを改変なしで複数同時に動作させることが可能となる．パフォーマンスを求められるような用途には不向きである一方，実行速度がそれほど重要ではなく，プロセストレース機能を利用したソフトウェアが提供する機能を利用したい場合は，本提案手法を適用することでそれらソフトウェアの組合せの制限を取り除くことができる．

6. 関 連 研 究

2.2 節で触れたように，UML には ptrace proxy と呼ばれる機構が実装されており，この機構を利用すると，すでにトレースされているプロセスでもデバッグすることが可能である．ptrace proxy では本提案手法と同様に，デバッグに必要なシグナルをデバッガに転送している．デバッガによる ptrace システムコールの処理は，ptrace proxy によりエミュレーションされている．また，プロセストレースを多重化できるのは UML 内のプロセスと UML カーネル自身のみである．本提案手法では，任意のアプリケーションどうしを連携させることが可能である．ptrace インタフェースを利用しているアプリケーションであれば，すべて多重化可能である．

7. お わ り に

本稿では，プロセストレース機能を持ったシステムコールを利用したソフトウェアを多重化する手法の提案を行った．我々の手法では，プロセスのトレースを専門に受け持つトレースマネージャを設けて，トレーサはトレースマネージャからプロセスの情報を取得する．

トレースマネージャの役割は大きく分けて 3 つあり，通知シグナルを受け取ること，受け取った通知シグナルを分配すること，既存インタフェースのエミュレーションである．通知シグナルを受け取るためには，プロセスを直接の子プロセスとすることで実現した．通知シグナルの分配では，トレーサ 1 つ 1 つに対して順番に通知を行い，監視動作を行わせる．すべてのモニタプロセスが実行継続を許可したときのみ，プロセスの実行が再開される．既存インタフェースのエミュレーションでは，プロセストレース機能を実現する ptrace システムコールの動作をエミュレーションすることで，既存のソフトウェアを改変せずに動作させることが可能である．

オーバヘッドはアプリケーションにより変化する．システムコール呼び出しの割合が多いプログラムはオーバヘッドが大きくなる．本提案手法の設計では，プロセストレース機構自身を利用してプロセストレース機構の動作を拡張しているため，元々のプロセストレース機構の動作速度が遅い問題をそのまま引き継いでしまっている．その代わり，プロセストレース機構が動作するシステムであれば本提案手法を適応可能であり，カーネルモジュールなどの実装方法に比べてバージョン依存も少なことが本提案手法の特徴である．今後の課題として，今回の実験では大幅なオーバヘッドが発生してしまうアプリケーションに対しても，小さいオーバヘッドで多重化プロセストレース機構を実現できる実装を検討したい．

謝辞　本研究は，平成 18 年度科学研究費補助金「安全なソフトウェア利用環境に関する研究」の支援を受けて行われた．また，本稿作成にあたり有益なご意見をいただいた鈴木勝博氏につつしんで感謝の意を表する．

参 考 文 献

1) 阿部洋丈，大山恵弘，岡　瑞起，加藤和彦：静的解析に基づく侵入検知システムの最適化，情報処理学会論文誌，Vol.45, No.SIG 3 (ACS 5), pp.11–20 (2004).
2) Dike, J: A user-mode port of the Linux kernel, *4th Annual Linux Showcase & Conference* (2000).
3) 大山恵弘，神田勝規，加藤和彦：安全なソフトウェア実行システム SoftwarePot の設計と実装，コンピュータソフトウェア，Vol.16, No.6, pp.2–12, 日本ソフトウェア科学会 (2002).
4) Albert, D.A., Maximilia, I, Klaus, E.S. and Chris, J.S.: UFO: A personal global file system based on user-level extensions to the operating system, *ACM Trans. Computer Systems* (*TOCS*), Vol.16, Issue. 3, pp.207–233 (1998).
5) Wagner, D. and Soto, P.: Mimicry Attacks on Host-Based Intrusion Detection Systems, *Proc. 9th ACM Conference on Computer and Communications Security*, pp.255–264 (2002).
6) Goldberg, I., Wagner, D., Thomas, R. and Brewer, E.A.: A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker, *Proc. 6th USENIX Security Symposium*, pp.1–13 (1996).

(平成 19 年 10 月 9 日受付)
(平成 20 年 2 月 4 日採録)

川崎　仁嗣（学生会員）

2006 年 3 月筑波大学第三学群情報学類卒業．同年 4 月より筑波大学大学院システム情報工学研究科コンピュータサイエンス専攻博士前期課程に入学，現在に至る．システムソフトウェア，特にコンピュータセキュリティに興味を持つ．

阿部　洋丈（正会員）

1999 年 3 月筑波大学第三学群情報学類卒業．2004 年 3 月筑波大学大学院博士課程工学研究科修了．博士（工学）．2004 年 4 月より科学技術振興機構戦略的創造研究推進事業 CREST 研究員．2007 年 3 月より豊橋技術科学大学情報工学系助手．同年 4 月より同助教．現在に至る．システムソフトウェア，特に分散システムとコンピュータセキュリティに興味を持つ．情報処理学会平成 16 年度山下記念研究賞，情報処理学会平成 16 年度論文賞受賞．日本ソフトウェア科学会，IEEE，ACM 各会員．

加藤　和彦（正会員）

1962 年生まれ．1985 年筑波大学第三学群情報学類卒業．1992 年博士（理学）（東京大学大学院理学系研究科）．1989 年東京大学理学部情報科学科助手，1993 年筑波大学電子・情報工学系講師，1996 年同助教授，2004 年筑波大学大学院システム情報工学研究科教授，現在に至る．オペレーティングシステム，セキュアコンピューティング，自律連合型分散システムに興味を持つ．