Vol. 42 No. 6 情報処理学会論文誌 June 2001

サーバのアクセス制限を安全に変更するための機構

光来健一[†] 千葉滋[††,†††]

インターネットサーバはつねにバッファオーバフロー攻撃などのクラック攻撃によって制御を奪われる危険にさらされている．クラック攻撃を防ぐ手法も研究されているが，サーバの制御が奪われるのはサーバ作成者のミスに起因するので，あらゆる攻撃を防げるようにするのは難しい．そこで，万一に備えてサーバにアクセス制限をかけて，クラック攻撃による被害を最小限に抑えることが必要である．実用的なサーバは様々なリクエストを処理しなければならないので，アクセス制限をかけるだけでなく，解除できるようにする必要もある．しかし，サーバはクラック攻撃されているかもしれないので，サーバが自分自身のアクセス制限を解除できるようにするのは危険である．本稿では，制御を奪われたサーバが不正にアクセス制限を解除できないようにするために，**プロセス・クリーニング**という手法を提案する．プロセス・クリーニングはサーバの制御を取り戻し，プロセスの状態を元に戻した後でのみ，アクセス制限の解除を許す．さらに，プロセス・クリーニングのオーバヘッドを減らすために，プロセスのメモリイメージを復元する方式をユーザが選択できるようにした．プロセス・クリーニングの性能について調べるために，我々の開発している Compacto オペレーティングシステム上で動く Apache ウェブサーバの性能を測定した．その結果，プロセス・クリーニングのオーバヘッドは許容できるものであった．

A Secure Mechanism for Changing Access Restrictions of Servers

Kenichi Kourai[†] and Shigeru Chiba[††,†††]

Internet servers are always in danger of being “hijacked” by various attacks like the buffer overflow attack. Although there are many researches to protect servers from such attacks, preventing all types of attacks is difficult because most of attacks are caused by programming errors of servers. Therefore, imposing access restrictions on servers is still needed so that it minimizes damages by hijacking. Since practical servers need various access restrictions to handle various requests, they have to not only impose but remove access restrictions. However, allowing hijacked servers to remove access restrictions is dangerous. In this paper, we propose our new technique called *process cleaning*, which prevents hijacked servers from illegally removing access restrictions. Process cleaning allows a process to remove access restrictions only after the thread of control is recovered and the state of the process is restored. To reduce the overhead of process cleaning, the users can select an implementation strategy for restoring the memory image of a process. We measured the performance of the Apache web server running on the Compacto operating system, which we have been developed. According to the results of our experiments, process cleaning can be implemented with acceptable performance overheads.

1. はじめに

ウェブサーバやメールサーバに代表されるサーバは，インターネットにおいてますます重要度を増している．このようなインターネットサーバは外部に対してサービスを提供するという性質上，つねにクラック攻撃の危険にさらされている．危険なコードをサーバに送り込み，サーバの制御を奪ってしまうバッファオーバフロー攻撃がその典型的な例である[18]．いったんサーバの制御が奪われると，攻撃者にそのサーバの権限を利用して不正行為を行われてしまう．このような攻撃を検出する様々な手法が提案されている[2),7),8),14)]が，クラック攻撃の原因はサーバプログラムのプログラミングミスや設計ミスであるため，あらゆる攻撃を防げるようにするのは難しい．

そこで，万一に備えてサーバにアクセス制限をかけておき，クラック攻撃による被害を最小に抑えること

† 東京大学大学院理学系研究科情報科学専攻
Department of Information Science, Graduate School of Science, University of Tokyo
†† 筑波大学電子・情報工学系
Institute of Information Science and Electronics, University of Tsukuba
††† 科学技術振興事業団さきがけ研究 21
PREST, Japan Science Technology Corp.

が必要である．しかし，実用的なサーバは様々なリクエストを処理しなければならないので，アクセス制限をかけるだけでなく，実行途中で解除する必要もある．しかし，サーバの制御が奪われている可能性があるので，サーバが自分自身のアクセス制限を解除できるようにするのは危険である．アクセス制限が不正に解除されるのを防ぐためには，サーバの制御が奪われているかどうかを判断しなければならないが，それは非常に難しい．たとえ正常に動いているように見えても，すでにクラックされていて，後でサーバの制御を奪えるように危険なコードが送り込まれているかもしれない．

本稿では，制御を奪われたサーバが不正にアクセス制限を解除するのを防ぐために，**プロセス・クリーニング**[20] という手法を提案する．プロセス・クリーニングはサーバを安全に保つために，プロセスからクラック攻撃の影響を取り除く．サーバがアクセス制限を解除する際には，サーバの制御が奪われている場合を考えて，まず，プロセスの制御を取り戻して，危険なコードの実行を終了させる．そして，メモリイメージを含むプロセスの状態をクラックされる前の状態に戻すことにより，サーバの中に隠された危険なコードを除去し，クラックされた実行環境を正常なものに戻す．この後でのみ，アクセス制限を安全に解除することができる．

また，本稿では，プロセス・クリーニングの実装の詳細とその最適化についても述べる．プロセス・クリーニングのオーバヘッドは，主にプロセスのメモリイメージの復元に起因する．このオーバヘッドを減らすために，サーバの作成者はメモリイメージを復元する方式をサーバの設計に合わせて選択することができる．また，サーバプログラムの静的データをうまく再配置することによっても，プロセス・クリーニングのオーバヘッドを減らすことができる．これらの最適化による性能改善を調べるために，我々の開発している Compacto オペレーティングシステム[20] 上で動く Apache ウェブサーバの性能を測定した．この実験結果を示し，プロセス・クリーニングのオーバヘッドについての議論を行う．

以下，2 章ではアクセス制限の解除にともなうセキュリティ上の危険について述べる．3 章ではアクセス制限の解除を安全に行うプロセス・クリーニングの手法について述べる．4 章ではプロセス・クリーニングの実装の詳細とオーバヘッドを減らす工夫について述べる．5 章ではプロセス・クリーニングのオーバヘッドを測定した実験について述べる．6 章で関連研究について述べ，最後に 7 章で本稿をまとめる．

2. アクセス制限の動的な変更

2.1 アクセス制限の解除の危険性

インターネットサーバは外部からクラック攻撃を受けて，サーバの制御を奪われる危険にさらされている．そこで，万一サーバの制御が奪われてしまった場合に備えて，サーバにアクセス制限をかけておき，クラック攻撃による被害を最小に抑えることが必要になる．我々の開発している Compacto オペレーティングシステムでは，プロセスに対してシステムコールレベルでアクセス制限を行うことができる．たとえば，クラックされたサーバがセキュリティ上重要なファイルを書き換えるのを防ぐために，サーバがそのようなファイルに対して `write` システムコールを発行するのをあらかじめ禁止することができる．また，UNIX でも `setuid` システムコールを使って，サーバを管理者権限よりも低いユーザの権限で動かすことができる．

実用的なサーバでは，アクセス制限をかけるだけでなくアクセス制限の解除も行えるようにし，アクセス制限を実行途中で変更できるようにする必要がある．たとえば，ウェブサーバがインターネットとイントラネットの両方からのリクエストを処理する場合がそうである．インターネットからの匿名ユーザによるリクエストを処理している間は，クラック攻撃に備えて `nobody` などの低い権限の下でサーバを実行すべきである．一方，イントラネット内部のユーザによるリクエストを処理している間は，クラック攻撃については考える必要はないが，そのユーザの権限の下でサーバを実行すべきであろう．たとえば，内部ユーザはウェブを通して自分しかアクセスできないファイルの内容も閲覧できた方がよい．

しかし，サーバがクラック攻撃を受けている可能性を考えると，サーバが自分自身のアクセス制限を解除できるようにするのは危険である．少なくとも，制御を奪われたサーバにはアクセス制限を解除させてはならない．たとえば，プロセスの実効ユーザ ID を変更する UNIX の `seteuid` システムコールはセキュリティホールになりうることが報告されている[13]．`seteuid` システムコールはサーバが自分自身に一時的にアクセス制限をかけ，後でそのアクセス制限を解除することを可能にする．アクセス制限の解除には何の制限もないので，途中でプロセスの制御を奪われてしまうと，不正にアクセス制限を解除されてしまう．

アクセス制限が不正に解除されるのを防ぐためには，サーバの制御が奪われているかどうかを判断しなけれ

ばならないが，それは非常に難しい．正常に動いているように見えるサーバであっても，後でサーバの制御を奪うような危険なコードが挿入されているかもしれない．このようなサーバにアクセス制限の解除を許すと，アクセス制限を解除した後で危険なコードを実行されてしまうかもしれない．サーバの中に危険なコードが入っていなくても，環境変数などサーバの実行環境がクラックされていることもありうる．たとえば，プログラムの引数 argv[0] が変更されていると，攻撃者がプロセスに HUP シグナルを送るだけで，任意のコマンドを実行できる場合がある[4]．隠されている危険なコードや実行環境のクラックを検出するのはきわめて難しい．

2.2 従来のサーバ構成法

サーバのアクセス制限を安全に解除できるようにするのは非常に難しいので，UNIX など従来の OS では，原則としてプロセスは一度かけたアクセス制限を解除できないようにしている．このためサーバは，受け取ったリクエストをアクセス制限をかけて処理し，後でそのアクセス制限を解除するということが直接にはできない．代わりに，子プロセスを作って，そのプロセスにアクセス制限をかけ，リクエストを処理させる必要がある．処理が終わった後は，必要ならば結果だけがプロセス間通信で親プロセス（サーバ本体）に渡され，子プロセス自体は終了させられる．たとえ子プロセスがクラックされて制御を奪われても，アクセス制限がかけられているので，被害は最小限に抑えられる．攻撃者は子プロセスのアクセス制限を解除して，システムを攻撃することはできない．結果を渡すためのプロセス間通信でクラックの影響が親プロセスに伝播しないかぎり，サーバ本体は安全である☆．また，次のリクエストを処理するためには，別の子プロセスが作られるので，仮に前の子プロセスがクラックされていても，その影響が次のリクエストの処理に伝播することはない．

子プロセスが，自分にかけられたアクセス制限の下では許可されない処理を実行する必要があるときは，その処理を実行できる別のプロセスに処理を依頼しなければならない．たとえば UNIX では，setuid ビットの立ったプログラムを子プロセスの子プロセス（孫プロセス）として起動し，処理を依頼する手法がしばしば使われる．setuid ビットが立ったプログラムは，起動したプロセスにかけられたアクセス制限ではなく，そのプログラムの所有者（たとえば root）にかけられたアクセス制限の下で実行される．仮に子プロセスがクラックされていても，プロセス間通信や環境変数を経由して影響が及ばない限り，孫プロセスは安全で，連鎖的にクラックされることはない．

このように，アクセス制限の解除を OS が許さないと，同等の効果を得るためには，プロセスの生成・破棄やプロセス間通信といったオーバヘッドがかかってしまう．しかし一般に，実用的なサーバではこのようなオーバヘッドは許容されない．子プロセスの生成は，サーバが複数のリクエストを並行して処理できるようにするためにも必要だが，実用的なサーバはプロセス生成のオーバヘッドを最小にするために，プロセスプールという手法を用いることが多い[9]．プロセスプールとは，あらかじめいくつかの子プロセスを作っておき，それらを繰り返し再利用してリクエストを処理させる手法である．このように処理速度を重視したサーバでは，リクエストを受け取るたびに子プロセスを生成するような手法は，たとえ安全性のためであっても採用しにくい．

3. プロセス・クリーニング

我々は，サーバが外部からクラック攻撃を受けて制御を奪われうる状況を想定し，万一の場合に備えて，可能なかぎりサーバにアクセス制限をかけられる Compacto オペレーティングシステムを開発している．アクセス制限をかけていると実行できない処理もあるので，実用上はアクセス制限の解除を行うことも必要であるが，それには 2 章で述べたように危険をともなう．我々は安全にアクセス制限の解除を行えるように，**プロセス・クリーニング**という機構を Compacto に実装した．プロセス・クリーニングを行えば 2.2 節で述べた，サーバが子プロセスを生成する手法ではなく，プロセスプールの手法を用いても，安全にアクセス制限を解除できるようになり，サーバの性能を改善することができる．

3.1 制御を奪われたプロセスの回復

安全にアクセス制限を解除するには，バッファオーバフロー攻撃などによって送り込まれた危険なコードからプロセスの制御を取り戻さなければならない．たとえまだ制御を奪われていないとしても，危険なコードがメモリに置かれているならばそれを除去しなければならない．そのために，安全なときのプロセスの状態を保存する save_state システムコールが用意されている．このシステムコールはプロセスの制御が奪われていないことが保証でき，かつ，アクセス制限がか

☆ プロセス間通信によってクラックの影響が伝播しないようにすることは必ずしも容易ではない．

```
save_state();                          (1)
accept();                              (2)
if (from_Internet)                     (3)
    ＜強いアクセス制限をかける＞
else
    ＜弱いアクセス制限をかける＞
＜リクエストの処理＞
restore_state();                       (4)
```

図 1 プロセス・クリーニングを使う典型的なサーバ
Fig. 1 A typical server using the process cleaning.

けられる前に呼ばれる．プロセスにかけられたアクセス制限は restore_state システムコールが呼ばれたときに解除される．このシステムコールはアクセス制限を解除し，同時にプロセスの状態を save_state システムコールで保存しておいた状態に戻す．復元されるプロセスの状態にはインストラクション・ポインタやメモリイメージも含まれるので，アクセス制限を解除するのと同時にプロセスの制御を取り戻し，メモリ上に置かれた危険なコードを除去することができる．

たとえば Compacto 上で動くウェブサーバは，**図 1** のようにこれら 2 つのシステムコールを使う．サーバの初期化が終わった後，ウェブサーバは save_state システムコールを呼び（図 1(1)），クライアントが接続してくるまで待つ（図 1(2)）．クライアントから接続されると，ウェブサーバは自分自身に適切なアクセス制限をかけて（図 1(3)），そのクライアントからのリクエストを処理する．この間では，たとえクラックされてサーバの制御が奪われても，被害は最小限に抑えられる．リクエストを処理し終わったら，ウェブサーバは restore_state システムコールを呼ぶ（図 1(4)）．このシステムコールはウェブサーバの状態を保存しておいた状態に戻す．インストラクション・ポインタも元に戻るので，サーバの制御は save_state システムコールの次の文（図 1(2)）に戻る．同時にサーバにかけられたアクセス制限も解除される．このようにして，ウェブサーバはクライアントからのリクエストを繰り返し処理する．

ウェブサーバの制御が奪われても，restore_state システムコールが呼ばれると制御を取り戻すことができる．攻撃者によって送り込まれたコードは restore_state システムコールを呼ばないかもしれないが，その場合にはアクセス制限を解除することはできず，望むような攻撃を行うことができない．この場合，送り込まれたコードに可能な攻撃はサービス拒否（DoS）攻撃だけである．DoS 攻撃は直接サーバの権限を濫用する攻撃ではないので，本研究の対象外である．ただし，Compacto が送り込まれたコードによる DoS 攻撃を検出できれば，OS カーネルは強制的に restore_state システムコールを実行して，サーバの制御を取り戻すことが可能である．たとえば，システムコールの発行の頻度が多すぎる場合や少なすぎる場合，または，多くのシステムコールがエラーを返している場合などに，DoS 攻撃が行われていると判断することができる．

3.2 プロセスの状態の保存・復元

save_state システムコールはシステムコールが呼ばれた時点でのプロセスの状態を保存する．保存されるプロセスの状態は，レジスタ，メモリイメージ，シグナルハンドラの内容，オープンされたファイルとソケットの情報である．これらの状態はカーネル空間に保存されるので，復元されるべき状態の安全性は保たれる．たとえユーザプロセスが攻撃を受けたとしても，保存されたプロセスの状態が攻撃者によって書き換えられることはない．また，同時に，このシステムコールが呼ばれた時点でプロセスにかけられているアクセス制限も記録する．

restore_state システムコールはすべてのレジスタの値を保存しておいた値に戻す．これは標準 C ライブラリで提供されている longjmp に似ている．元に戻されるレジスタの中にはインストラクション・ポインタも含まれる．送り込まれた危険なコードが首尾よくプロセスの制御を手に入れられたとしても，その実行は restore_state システムコールが呼ばれた時点で終了させられる．危険なコードは自ら save_state システムコールを呼び出して，自分が制御を得た状態を保存しようとするかもしれない．しかし，その時点でのすべての状態が保存されるので，その後の restore_state システムコールでは以前にかけられたアクセス制限を解除することはできなくなる．

restore_state システムコールはメモリイメージを保存しておいた内容に戻す．これによりメモリ上に残されて後で実行される危険なコード（トロイの木馬）を除去する．さらに，メモリ上の環境変数などの実行環境も save_state システムコールが呼ばれたときの状態に戻される．不正メモリアクセスと通常のメモリアクセスとを区別するのは非常に難しいので，メモリイメージ全体を元に戻す．また，メモリイメージを保存しておくことで，サーバを安定に保つという副次的な効果を得ることができる．サーバがクラック攻撃されると，その成否にかかわらず，サーバのメモリが破壊されて異常終了する原因になるかもしれない．この

ような場合でも OS カーネルがメモリフォールトを検出したときに，破壊されたサーバのメモリを修復すれば，サーバの実行を継続することができる．

restore_state システムコールはシグナルハンドラを保存しておいたハンドラに戻す．シグナルハンドラが不正なハンドラに置き換えられてしまうと，アクセス制限が解除された後で攻撃者がプロセスにシグナルを送ることにより，不正なハンドラを実行できてしまう．シグナルハンドラの復元はこのような不正なハンドラがサーバの制御を奪うのを防ぐ．もし未処理のシグナルがあれば，シグナルハンドラを元に戻す前に処理する．これにより未処理のシグナルも正しいシグナルハンドラによって処理される．

restore_state システムコールは save_state システムコールが呼ばれた後でオープンされたファイルやソケットをクローズする．これにより不正に張られたネットワーク接続などを切断することができる．逆に，攻撃者によってファイルやソケットが不正にクローズされていれば，それらを再オープンして対応するファイル記述子を元に戻す．これはオープンされているはずのファイルがクローズされて，サーバの実行が妨害されるのを防ぐためである．

このような状態の復元を行った後で，restore_state システムコールはプロセスのアクセス制限を解除し，save_state システムコールで記録された元のアクセス制限の状態に戻す．

3.3 プロセス・クリーニングの制限

プロセス・クリーニングは restore_state システムコールによってプロセスの状態を完全に元に戻すので，アクセス制限を解除する前後で情報を受け渡すことができない．そのため，個々のリクエスト処理を独立に行うことができるウェブサーバのようなサーバでのみ使うことができる．サーバの中にはリクエストの処理が終わった後もクライアントの情報を保持したり，データをキャッシュしたりするものもある．プロセス・クリーニングはセキュリティ上の理由から，このような情報をプロセス内に残すことを許していない．

4. プロセス・クリーニングの実装と最適化

プロセス・クリーニングを実用的なサーバで使えるようにするには，少なくとも子プロセスを作る従来の手法に比べて高速でなければならない．しかしながら，プロセス・クリーニングを単純に実装すると，子プロセスを作る場合と同等か，それ以上のオーバヘッドがかかる．そこで，本章ではプロセス・クリーニングを高速に実行できるようにする実装上の工夫について述べる．

4.1 メモリイメージの保存・復元

プロセス・クリーニングのオーバヘッドは主に，プロセスの状態を保存・復元するためのメモリコピーに起因する．save_state システムコールでメモリイメージ全体を保存し，restore_state システムコールで全体を復元するという単純な実装では，まったく変更されていないメモリに関してもコピーが起こるのでオーバヘッドが大きい．そこで我々は，メモリのコピー量を減らすために，コピー・オン・ライト[3),15)] の手法を用いている．さらに，ユーザはメモリイメージを復元する方式を選択することができる．図 1 に示されているように，Compacto 上の典型的なサーバは状態を最初に 1 回だけ保存し，リクエストの処理が終わるたびに保存しておいた状態に戻す．ユーザはこのような場合に効率良くプロセス・クリーニングを行える方式を選択することができる．

save_state システムコールは状態を復元できるようにプロセスのメモリイメージを保存する．ただし，このシステムコールが呼ばれた時点ですべてのメモリイメージを保存するわけではない．まず，すべての書き込み可能なメモリページを書き込み禁止にする．プロセスがそのページに書き込もうとしてページフォールトが起きたとき，そのページの内容を保存するためにページが複製される．そしてページテーブルを書き換えることで，元のページはカーネル空間に移動され，複製によって割り当てられた新しいページがその仮想アドレスにマップされる☆．たとえユーザプロセスの制御が奪われたとしても，元のページはカーネル空間に保存されているので，不正に書き換えられることはない．この新しいページをシャドウページと呼ぶ．シャドウページは書き込み可能になっており，以降の書き込みではページフォールトは起きない．

restore_state システムコールは書き込みが行われたメモリページだけを元の内容に戻す．メモリページの内容を元に戻すために，ユーザは 2 種類の方式から選択することができる．1 つはシャドウページをアンマップして破棄し，カーネル空間に保存されていた元のページを元の仮想アドレスに戻す方法である（**図 2** (a)）．これを再マップ方式と呼ぶ．もう 1 つは元のページの内容をシャドウページにコピーする方法である．元のページはカーネル空間に残される（図 2 (b)）．これをコピー方式と呼ぶ．

☆ 元のページは親プロセスまたは子プロセスとコピー・オン・ライトの状態で共有されているかもしれないので，元のページの方を保存しておかなければならない．

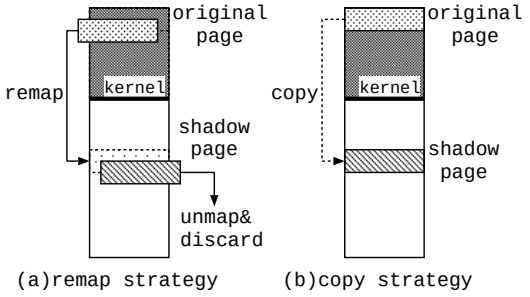


図 2 メモリイメージを復元する 2 つの方式
Fig. 2 Two strategies for restoring memory.

再マップ方式はメモリイメージを復元する際にメモリをコピーする必要がないので，デフォルトではこの方式が選択される．この方式にはメモリの消費量を抑えるという利点もある．ただし，ユーザはコピー方式を使うように指定することもできる．Compacto 上の典型的なサーバは `save_state` システムコールで保存された状態を繰り返し復元するので，`save_state` システムコールと `restore_state` システムコールの発行が 1 対 1 に対応しない．このような場合，再マップ方式の方が効率が悪くなる可能性がある．再マップ方式を使うと，`restore_state` システムコールで復元されたメモリページは，次の書き込みを検出するために再び書き込み禁止にされる．そしてページフォールトが起こったときには，シャドウページを割り当て，元のページの内容をコピーしなければならない．

一方，コピー方式ではシャドウページがマップされたまま再利用され，元のページはカーネル空間に保存されている．これらのページに対しては，書き込み禁止にしてページフォールトを起こす必要はない．それゆえ，コピー方式は再マップ方式と比べてページフォールトの回数が少なくて済む．ただし，メモリコピーの量は次に `restore_state` システムコールが呼ばれるまでのメモリアクセスパターンに依存する．プロセスがつねにまったく同じメモリページ，つまり，シャドウページだけに書き込みを行えば，コピー方式は再マップ方式と同じ量のメモリコピーだけで済む（**図 3** (a),(b1)）．この場合，ページフォールトが起こらない分だけ，コピー方式の方が再マップ方式より速くなる．一方，プロセスが異なるメモリページに書き込みを行った場合は，コピー方式の方がメモリコピーの量が増えてしまう．`restore_state` システムコールでのシャドウページへのコピー（図 3 (a)）の一部は無駄になり，ページフォールトが起こったときには別のページを複製しなければならない（図 3 (b2)）．

コピー方式が選択されたとき，`restore_state` システムコールは複製されているメモリページのうち，ダーティビットが立っているページだけを復元する．ダーティビットはプロセスが対応するメモリページに書き込みを行ったときに，ハードウェアによってセットされる．メモリイメージを復元した後，シャドウページへの次の書き込みを判定できるようにするために，すべてのダーティビットをリセットする．`restore_state` システムコールはダーティビットが立っていないシャドウページについては復元のためのコピーを行わない．

コピー方式はいったん複製されたメモリページに対して，つねに元のページとシャドウページを保持しておかなければならないので，再マップ方式よりも多くのメモリを必要とする．メモリ消費量を抑えるために，ユーザは使われていないページの複製をやめさせることができる．各ページのダーティビットの履歴を調べることで，ページがどのくらいの間使われていないかを知ることができる．`restore_state` システムコールは複製をやめるページのシャドウページをアンマップして破棄し，カーネル空間にある元のページを元の仮想アドレスに戻す．そのページは次の書き込みを検出するために書き込み禁止にされる．

4.2 サーバプログラムの最適化

サーバプログラムを標準 C ライブラリのような必要なすべてのライブラリと静的にリンクすることによって，プロセス・クリーニングのオーバヘッドを減らすことができる． ライブラリが動的にリンクされていると，ライブラリの保持している静的データはプログラムのものとは別の領域に配置される．一方，静的にリンクされていれば，すべての静的データは同じ領域に配置される．このため，静的データがメモリ上に連続して配置されるので，必要とされるメモリページ数を減らすことができる．その結果，`restore_state` システムコールで復元されなければならないページ数も減らすことができる．また，この最適化は動的リンクにともなうオーバヘッドも減らすことができる．

さらに，サーバ作成者は，復元されるメモリページ数を減らすために我々が開発したツールを使うことができる．このツールは，メモリへの書き込みの局所性が増すように，実行時のプロファイル情報を使って静的データの再配置を行う．まず，サーバプログラムにメモリ書き込みを記録するためのコードを挿入し，そのサーバプログラムを動かしてプロファイル情報を得る．そしてこの情報に基づいて，`restore_state` システムコールによって復元されるメモリページ数が最小になるように，静的データを再配置する．実際のアクセスパターンが集めたプロファイル情報とうまく一致

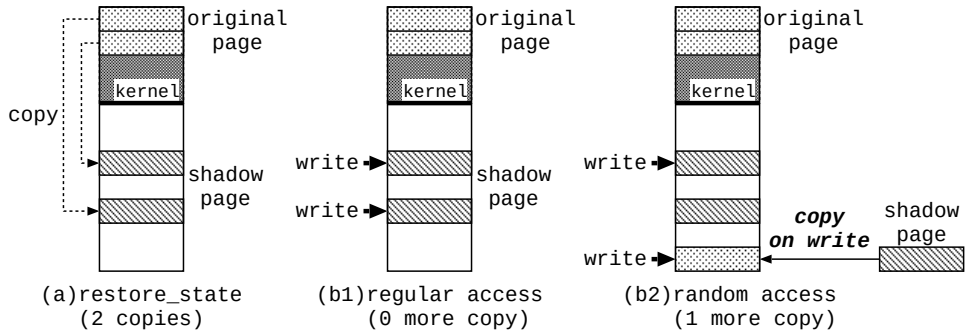


図 3 コピー方式で起こりうるメモリコピーの様子
Fig. 3 Possible memory copies in the copy strategy.

表 1 save_state システムコールの実行時間
Table 1 The execution time of the save_state system call

ページ数	12	16	32	64	128	256	512	1024
μsec	5.29	5.62	6.34	7.91	11.4	18.1	28.8	56.1

すれば，サーバの性能を改善することができる．静的データの再配置により，メモリイメージを保存する際に発生するページフォールトの回数も減らすことができる．さらに，1 つのキャッシュラインの中の多くのデータが実際に使われるので，キャッシュのヒット率が改善される可能性もある．

5. 実　　験

我々は Linux 2.2.16 カーネルをベースにして Compacto オペレーティングシステムを開発した．本節ではプロセス・クリーニングのオーバヘッドを測定するために行った実験について述べる．実験に用いたマシンは CPU が Pentium III 933MHz☆，メモリが 256MB の PC である．

5.1 マイクロベンチマーク

まず，save_state システムコールと restore_state システムコールの実行時間を測定した．save_state システムコールの実行時間はプロセス内の書き込み可能なメモリページの数に依存する（**図 1**）．参考値として，Apache ウェブサーバでは約 40 ページが書き込み可能になっている．全実行時間はメモリページ以外のリソースの状態を保存するコスト（3.4μsec）を含む．そのうち，2.12μsec はシグナルハンドラの保存，0.90μsec はオープンされたファイルやソケットの状態の保存，0.05μsec はレジスタの保存，0.33μsec はシステムコールの呼び出しにかかるコストであった．

☆ L1 キャッシュ 16KB（命令）+16KB（データ）．L2 キャッシュ256KB.

表 2 リソースの状態を復元するコスト（μsec）
Table 2 The cost of restoring the states of resources (μsec).

リソース数	0	1	2	4	8	16	32
シグナルハンドラ	0.08	0.27	0.33	0.38	0.60	1.05	N/A
ファイル・ソケット	0.05	0.60	0.69	0.83	1.05	1.49	2.84

ファイルやソケットの数による影響はほとんど無視できる範囲であった．

restore_state システムコールを実行するのには，システムコールを呼び出すのに 0.33μsec，レジスタを復元するのに 0.05μsec かかった．シグナルハンドラとファイル・ソケットの復元にかかるコストは**図 2** に示されているように，復元されるリソースの数に依存する．メモリイメージの復元にかかるコストも復元されるページ数に依存する（**図 4**）．restore_state システムコールでは，再マップ方式の方がコピー方式より小さいコストで済んでいるように見える．しかし，再マップ方式では，復元したメモリページに再び書き込みが行われるとページフォールトが起こるので，余分なコストがかかる．このため，最悪の場合，再マップ方式はコピー方式の 1.3〜1.8 倍のコストがかかる．

5.2 Apache ウェブサーバ

我々は WebStone ベンチマークプログラム[11] を用いて，Compacto 上で動くウェブサーバの性能を測定した．ウェブサーバとして，プロセスプールの手法を用いて実装されている Apache 1.3.12[1] を用いた．クライアントマシンとして，CPU が Celeron 300MHz，メモリが 64MB の PC を最大 16 台用いた．クライアントマシンの OS は FreeBSD 3.4 であった．ネットワークの飽和を避けるため，サーバマシンにはイーサネットポートを 2 つ用意し，クライアントマシンとサーバマシンは 100baseT のイーサネットで接続した．

比較のため，4 種類の Apache ウェブサーバを用いた．POOL サーバはプロセス・クリーニングを行わな

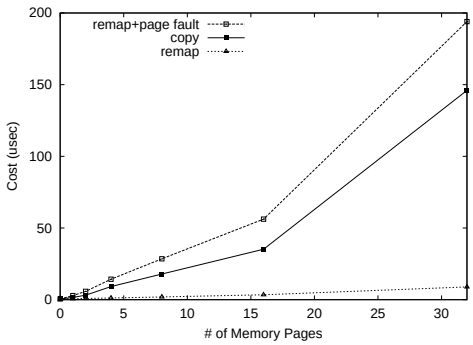


図 4 メモリイメージ復元のコストの比較
copy と *remap* は `restore_state` システムコールにおけるコスト．*remap*+*page fault* は再マップ方式の最悪の場合のトータルコスト

Fig. 4 Comparison of the cost of memory restoration. *copy* and *remap* are the cost of memory restoration in the `restore_state` system call. *remap*+*page fault* is the total cost of the remap strategy in the worst case.

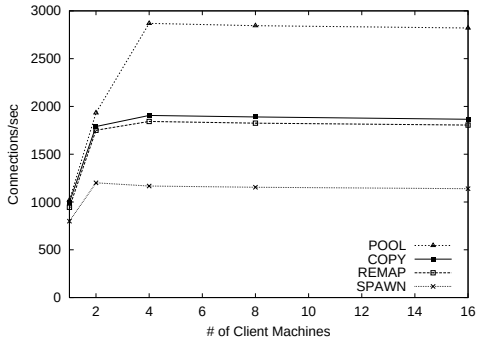


図 5 0 バイトファイルのリクエストに対するサーバのスループット
Fig. 5 The server throughput (0 byte file was requested)

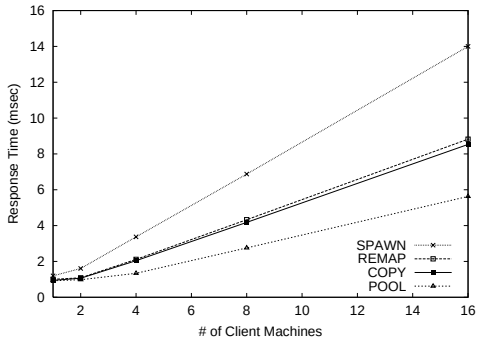


図 6 0 バイトファイルのリクエストに対する応答時間
Fig. 6 The response time (0 byte file was requested)

いサーバである．COPY サーバと REMAP サーバはプロセス・クリーニングを行い，それぞれコピー方式と再マップ方式を使うサーバである．これらの 3 種類のサーバは 16 個のプロセスをあらかじめ作っておく．また，SPAWN サーバはリクエストごとに子プロセスを作るように変更を加えた Apache ウェブサーバである．このサーバはプロセスプールの手法を用いず，プロセス・クリーニングも行わない．この実験ではどのサーバに対してもアクセス制限をかけなかった．

5.2.1 実験結果

図 5 と**図 6** はクライアントからサイズが 0 バイトの HTML ファイルを繰り返しリクエストされた場合の，サーバのスループット（1 秒間に処理したリクエストの数）と応答時間を示している．プロセス・クリーニングで復元されるメモリページは 8 ページ，シグナルハンドラは 1 つ，ファイルとソケットが 1 つずつであった．

図 7 と**図 8** はクライアントから様々なファイルのリクエストが行われた場合の，サーバのスループットと平均応答時間を示している．プロセス・クリーニングで復元されるメモリページは平均 8.1 ページ，シグナルハンドラは 1 つ，ファイルとソケットが 1 つずつであった．リクエストされるファイルは HTML ファイル，バイナリファイル，CGI プログラムによって生成されるファイルであり，我々が実際に使用しているウェブサーバ☆からコピーしたものである．リクエストされたファイルサイズの平均は 7.6KB（73B〜47KB）であった．

☆ `http://www.hlla.is.tsukuba.ac.jp/`

5.2.2 考　　察

実験結果から，COPY サーバは SPAWN サーバに対して平均で 50%性能が良くなっていることが分かる．従来，サーバがリクエストに応じてアクセス制限をかけるにはリクエストごとに新しい子プロセスを作らなければならなかった．プロセス・クリーニングを用いたサーバはこのような従来型のサーバと同程度の安全性を保ちながら，十分な性能改善を達成している．プロセス・クリーニングのコストは POOL サーバと比較すると最大で 35%と決して小さくはないが，プロセスプールを用いたサーバのセキュリティを確保することができるという十分な利点がある．

プロセス・クリーニングは CPU に負荷をかけるので，サーバの実行の多くがディスク I/O やネットワーク I/O に費やされる場合，プロセス・クリーニングのオーバヘッドは相対的に小さくなる．これはプロセス・クリーニングの実行が I/O によって隠蔽されやすくなるからである．実際，様々なファイルのリクエ

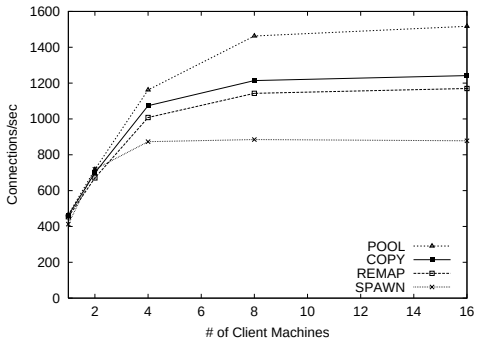


図 7 様々なファイルのリクエストに対するサーバのスループット
Fig. 7 The server throughput (various sizes of files were requested).

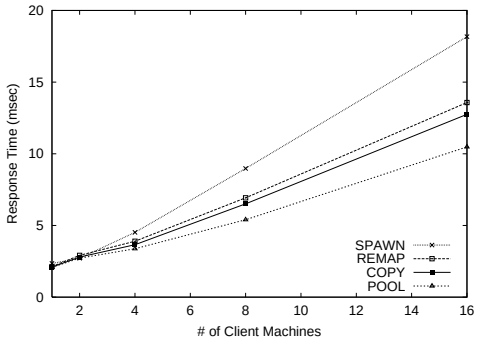


図 8 様々なファイルのリクエストに対する平均応答時間
Fig. 8 The average response time (various sizes of files were requested).

ストが行われた場合の方が 0 バイトファイルのリクエストの場合より大きなファイルを扱うため，ディスク I/O とネットワーク I/O が増え，COPY/REMAP サーバの性能が良くなっている．

実験結果から，COPY サーバはつねに REMAP サーバより約 5%性能が良くなっていることが分かる．0 バイトの同一の HTML ファイルがリクエストされた場合，サーバがリクエストを処理するときにはまったく同じメモリページに書き込みが行われる．これはコピー方式が最も良い性能を示す場合である．一方，様々なリクエストが行われた場合，全く同じリクエストを繰り返した場合に比べてコピー方式の利点が多少失なわれるが，それでもコピー方式の方が良い性能を示した．Apache ウェブサーバの場合にはコピー方式は再マップ方式よりも良い性能を示したが，他のサーバの場合には再マップ方式の方が良い性能を示すことも考えられる．

COPY サーバは REMAP サーバよりも多くのメモリを必要とするので，メモリ消費量を抑えるために，使われていないページの複製をやめることもできる．我々はこの方式の効果についても測定を行った．この実験では，`restore_state` システムコールが呼ばれた時点でダーティビットが立っていないシャドウページを使われていないページと見なした．実験によると，この修正を加えた COPY サーバは様々なリクエストが行われた場合には，平均 4.9 ページのメモリ消費を抑えることができた．ただしその分だけページフォールトの回数が増加し，元の COPY サーバと比べると性能が 1%低下した．

これらの実験結果はサーバプログラムの静的データを最適に再配置したものについて測定した結果である．静的データを再配置することにより，Apache ウェブサーバでは，静的データの変更によって書き込みが行われるメモリページが 10 ページから 1 ページへと減少した．この結果，プロセス・クリーニングを行う COPY/REMAP サーバは平均で 40%性能が良くなった．一方，SPAWN サーバでも約 40%性能が良くなった．これは SPAWN サーバでも子プロセスを作る際にコピー・オン・ライトの技術が使われるため，REMAP サーバの場合と同じだけ，メモリコピーとページフォールトが減るからである．また，POOL サーバでも 4%程度性能が良くなったが，これはライブラリを動的リンクすることによるオーバヘッドなどが減少したためと考えられる．

5.3 FastCGI

我々は Apache ウェブサーバ上で動く FastCGI モジュール[16] の実行性能についても測定を行った．FastCGI モジュールは CGI プログラムを動かすのにプロセスプールの手法を用いる．FastCGI モジュールを使わない場合，サーバは CGI プログラムを動かすたびに子プロセスを作って実行しなければならない．CGI プログラムには最もよく使われているアクセスカウンタの 1 つである wwwcount 2.5[12] を用いた．

この実験でも POOL，COPY，REMAP，SPAWN の 4 種類のサーバについて実験を行った．COPY/REMAP サーバで用いられる FastCGI モジュールだけがプロセス・クリーニングを行う．SPAWN サーバは FastCGI モジュールを用いず，CGI プログラムを動かすたびに子プロセスを作る．それぞれのウェブサーバ本体にはプロセスプールを行う通常の Apache を用いた．

図 9 と**図 10** に実験結果を示す．この結果は Apache ウェブサーバの場合と似ている．しかし，プロセス・クリーニングのオーバヘッドはかなり小さくなっており，COPY サーバは POOL サーバに比べて 8%性能が悪くなっているだけである．これは CGI プログラムの実行自体がボトルネックになり，プロセス・クリー

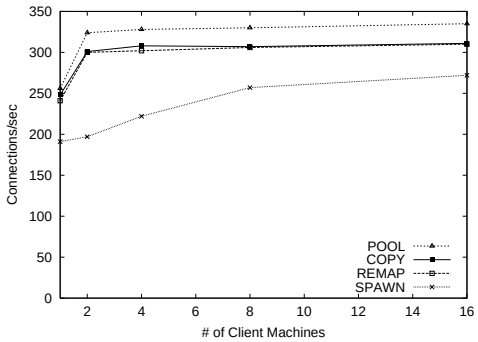


図 9 CGI プログラムのリクエストに対するサーバのスループット
Fig. 9 The server throughput (CGI program was requested).

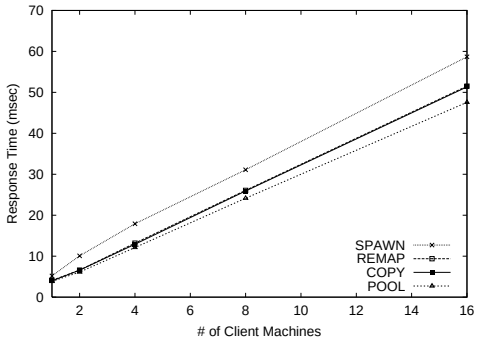


図 10 CGI プログラムのリクエストに対する平均応答時間
Fig. 10 The average response time (CGI program was requested).

ニングのオーバヘッドが相対的に小さくなったためである．

6. 関 連 研 究

プロセス・クリーニングはチェックポインティングによる回復の一種であると考えることができる．コピー・オン・ライトを使って効率良くチェックポインティングを実装した研究もある[5),10)]．我々はこの手法をフォールトトレラントなどの従来の領域の代わりに，セキュリティの領域に適用した．ただし，プロセス・クリーニングの設計はアクセス制御機構に特化されている．たとえば，変更されたメモリページだけが保存され，保存されたページはディスクに書き出されない．

我々はプロセス単位でのアクセス制御を考えたが，プロセスの代わりにスレッドが使われることもある．しかし，スレッドの間にはリソースに対する保護がないので，1 つのスレッドの制御が奪われると全てのスレッドに影響が及ぶ．そこで，プロセスという従来の保護ドメインの中に複数の軽量な保護ドメインを作る研究が行われている[6),17)]．ただしこの研究の目的は，プログラム・モジュールをプロセス内の保護ドメインに割り当て，その間の切替えを高速にすることである．スレッドに保護ドメインを割り当てることも可能だが，そのオーバヘッドがどの程度になるかは不明である．また，OS のサポートなしで軽量な保護ドメインを作る手法も提案されている[19)]．この手法はプログラムのメモリアクセスをチェックすることにより保護ドメインを作る．しかし，外部から不正に送り込まれたコードにはチェックが及ばず，保護ドメインの中で実行させることができない．

また，バッファオーバフロー攻撃そのものを防ぐために，様々な手法が提案されている．StackGuard[7)] はバッファ用メモリの隣りに特殊な値を書き込んでおき，その値の変化の有無でバッファオーバフローを検出する．libsafe[2)] は標準 C ライブラリに対して透過的にバッファの境界チェックを行う．また，スタックを実行禁止にすることで，スタックオーバフローで送り込まれた危険なコードを実行できないようにする方法も提案されている[14)]．これらの方法とプロセス・クリーニングを組み合わせて使い，クラック攻撃を検出したらプロセスを回復して，即座に実行を再開させることもできる．

7. ま と め

本稿では，制御を奪われたサーバが不正にアクセス制限を解除するのを防ぐプロセス・クリーニングという手法を提案した．さらに，プロセス・クリーニングの 2 種類の実装方式として，再マップ方式とコピー方式を示した．我々の行った実験によると，プロセス・クリーニングによるオーバヘッドは最悪の場合には 35% 程度になった．その代わり，クラック攻撃に対して脆弱なプロセスプールを用いたサーバを十分安全にできている．また，リクエストごとに子プロセスを作る従来のサーバと比べると，同程度の安全性を保ったまま，平均で 50%性能を改善することができた．

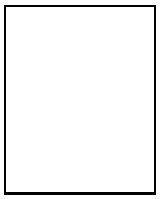
謝辞 研究に関して適切な助言をくださった筑波大学の板野肯三先生，立堀道昭氏をはじめとする HLLA 研究室の方々，東京大学の清水謙多郎先生，および本稿の執筆にあたり有益な助言をくださった査読者の方々に感謝いたします．

参 考 文 献

1) Apache HTTP Server Project: Apache HTTP Server, `http://www.apache.org/`.
2) Baratloo, A., Tsai, T. and Singh, N.: Transparent Run-Time Defense Against Stack

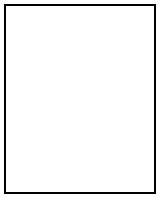
Smashing Attacks, *Proc. USENIX Annual Technical Conference* (2000).
3) Bobrow, D. G., Burchfiel, J. D., Murphy, D. L. and Tomlinson, R. S.: TENEX, a Paged Time Sharing System for the PDP-10, *Comm. ACM*, Vol. 15, No. 3, pp. 1135–1143 (1972).
4) CERT: Sendmail Daemon Mode Vulnerability, CERT Advisory CA-96.24.
5) Cheriton, D. R. and Duda, K. J.: Logged Virtual Memory, *Proc. 15th ACM Symposium on Operating Systems Principles*, pp. 26–39 (1995).
6) Chiueh, T., Venkitachalam, G. and Pradhan, P.: Integrating Segmentation and Paging Protection for Safe, Efficient and Transparent Software Extensions, *Proc. the 17th ACM Symposium on Operating Systems Principles*, pp.140–153 (1999).
7) Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q. and Hinton, H.: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks, *Proc. the 7th USENIX Security Symposium*, pp. 63–78 (1998).
8) Cowan, C., Wagle, P., Pu, C., Beattie, S. and Walpole, J.: Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade, *Proc. DARPA Information Survivability Conference and Expo* (2000).
9) Hu, J., Mungee, S. and Schmidt, D.: Principles for Developing and Measuring High-Performance Web Servers over ATM, Technical Report 97-09, Department of Computer Science, Washington University (1997).
10) Li, K., Naughton, J. F. and Plank, J. S.: Real-Time Concurrent Checkpoint for Parallel Programs, *Proc. 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 79–88 (1990).
11) Mindcraft: WebStone Benchmark, `http://www.mindcraft.com/webstone/`.
12) Muquit, M. A.: WWW Homepage Access Counter and Clock, `http://www.muquit.com/muquit/software/Count/Count.html`.
13) NetBSD Security Alert Team: at(1) vulnerabilities, NetBSD Security Advisory NetBSD-SA1998-004.
14) Openwall Project: Non-Executable User Stack, `http://www.openwall.com/linux/`.
15) Rashid, R., Tevanian, A., Young, M., Golub, D. and Baron, R.: Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures, *Proc. 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 31–39 (1987).
16) Saccoccio, R.: FastCGI, `http://www.fastcgi.com/`.
17) Takahashi, M., Kono, K. and Masuda, T.: Efficient Kernel Support of Fine-grained Protection Domains for Mobile Code, *Proc. 19th IEEE International Conference on Distributed Computing Systems*, pp. 64–73 (1999).
18) Wagner, D., Foster, J., Brewer, E. and Aiken, A.: A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities, *Proc. Network and Distributed Systems Security Symposium*, pp. 3–17 (2000).
19) Wahbe, R., Lucco, S., Anderson, T. E. and Graham, S. L.: Efficient Software-Based Fault Isolation, *Proc. 14th Symposium on Operating Systems Principles*, pp. 203–216 (1993).
20) 光来健一, 千葉滋: 動的なアクセス権限変更のためのアクセス制限の安全な解除機構, 情報処理学会研究報告，2000-OS-85, pp. 55–62 (2000).

(平成 12 年 12 月 15 日受付)
(平成 13 年 4 月 6 日採録)



光来 健一（学生会員）

1975 年生．1999 年東京大学大学院理学系研究科情報科学専攻修士課程修了．現在，同大学院理学系研究科情報科学専攻博士課程在学中．オペレーティングシステムの安全性に関する研究に従事．ACM 会員．



千葉 滋（正会員）

1968 年生．1996 年東京大学大学院理学系研究科情報科学専攻博士課程退学，同専攻助手．1997 年より筑波大学電子・情報工学系講師，2001 年より東京工業大学大学院情報理工学研究科数理・計算科学専攻講師．理学博士．プログラミング言語およびオペレーティング・システムに興味を持つ．日本ソフトウェア科学会，ACM 各会員．日本ソフトウェア科学会高橋奨励賞，同会論文賞受賞．