Vol. 39 No. 11 情報処理学会論文誌 Nov. 1998

多段階保護機構：拡張可能 OS の新しい Fail-safe 機構

光来健一[†] 千葉滋[††] 益田隆司[†]

拡張可能 OS は拡張モジュールを追加することで，その機能を動的に拡張することができる．このような OS では，拡張モジュールのエラーから OS を守るために，fail-safe 機構が必要とされる．しかしながら，十分な fail-safe 機構を実現しようとすると，従来の実装技術ではシステムの性能低下が避けられなかった．そこで我々は，新しい fail-safe 機構である多段階保護機構を提案する．この機構は拡張モジュールを，変更なしに，様々な保護レベルで OS に組み込むことを可能にする．この機能を用いて，デバッグ時には保護レベルを高くし，デバッグが終われば保護レベルを低くするなど，必要最低限の保護レベルで拡張モジュールを動かすことにより，システムの性能低下を回避することができる．我々は，ファイルシステムに対して多段階保護機構を実現するシステムを NetBSD 1.2 上に実装した．このシステムでは，保護マネージャと呼ばれるものが複数用意されており，それぞれが異なる保護レベルを提供している．そしてユーザは保護マネージャを交換することで，拡張モジュールの保護レベルを変更することができる．さらに，我々は多段階保護機構の有効性を調べるために実験を行い，fail-safe の機能を減らせば，その分実行性能を改善できることを確かめた．また保護レベルが最大のときでも，そのオーバヘッドは実用に耐える程度であり，保護レベルを最小にすれば，カーネルに作り込んで手で最適化したファイルシステムに十分近い性能が得られることも分かった．

Multi-level Protection: A New Fail-safe Mechanism for Extensible Operating Systems

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

Extensible operating systems enable the users to extend their functions by adding extension modules on demand. Such operating systems need a fail-safe mechanism for protecting the systems from erroneous extension modules. However, this mechanism with full protection capability has implied serious performance penalties. To address this problem, we propose a new fail-safe mechanism called multi-level protection. It allows the users to install an extension module in the operating system at various protection levels without changing the source code, and thereby, the users can run the module at the minimal protection level to avoid performance penalties. For example, they can choose a higher level for an unstable module, but a lower one for a stable module. We have implemented the multi-level protection for file systems of NetBSD 1.2. This system provides multiple protection managers of various protection levels so that the users can choose one of the protection managers and easily change the protection level. We confirmed that the performance of the file system modules is improved if the protection level is lower, that the overheads at the maximal protection level are not too large, and that the overheads at the minimal protection level are almost negligible compared with those of the file system hand-crafted in the kernel.

1. はじめに

コンピュータが様々な用途に使われるようになり，ユーザ層の拡大にともなって，OS に対するユーザの要求が多様化してきている．しかし，それらの様々な要求を満たすような汎用 OS を提供しようとすると，OS の肥大化を招いてしまう．そのうえ，すべての要求を満たすことは不可能である．そこで，後から OS の機能を拡張するためのプログラムを OS に追加することで，より広くユーザの要求に答えられるようにすることが考えられるようになった．

このような OS は拡張可能 OS と呼ばれ，拡張モジュールと呼ばれるプログラムを OS に追加することによって，OS の機能を拡張することができる．このような OS の重要な機構の 1 つに，拡張モジュールのエラーから OS を守る fail-safe 機構がある．fail-safe 機構は

† 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

拡張モジュールのエラーを検出すると，そのエラーがOS の他の部分に悪影響を及ぼさないようにする．しかし十分なfail-safe 機構を実現しようとすると，従来の実装技術ではシステムの性能低下が避けられなかった．

そこで我々は，新しいfail-safe 機構である多段階保護機構[9] を提案する．この機構は拡張モジュールを変更なしに，様々な保護レベルでOS に組み込むことを可能にする．この機能により，ユーザは保護レベルを変更して，不要なfail-safe 機構によるシステムの性能低下を回避することができる．比較的安定している拡張モジュールの場合には，簡略化したfail-safe 機構を用いて性能向上をはかり，逆に不安定な拡張モジュールの場合には，性能を犠牲にしても完全なfail-safe 機構を用いた方がよい．

我々は，ファイルシステムに対して多段階保護機構を実現するシステムをNetBSD 1.2 上に実装した．このシステムでは，保護マネージャと呼ばれるものが複数用意されており，それぞれが異なる保護レベルを提供している．そしてユーザは保護マネージャを交換することで，拡張モジュールの保護レベルを変更することができる．保護マネージャは，検出できるエラーや回復できるエラーを変えることによって，異なる保護レベルを提供している．また，保護マネージャは拡張モジュールのためのApplication Programming Interface（API）を提供しており，保護レベルの変更の際に，拡張モジュールのソースコードを書き換える必要はない．

我々は多段階保護機構の有効性を調べるために，いくつかの実験を行った．その目的は第1 に， fail-safe の機能を減らせば，その分実行性能を改善できるかどうかを調べることである．第2 に，保護レベルが最大のときにそのオーバヘッドがどの程度であるかを調べることである．そして第3 に，保護レベルを最小にしたときに，カーネルに作り込んで手で最適化したファイルシステムと比べて，どの程度の性能が得られるかを調べることである．これらの実験から，ファイルシステムに対して多段階保護機構は有効に機能することが確かめられた．

以下， 2 章では拡張モジュールのエラーからOS を守るfail-safe 機構について説明し，これまでの拡張可能OS のfail-safe 機構の概観を述べる． 3 章では我々の提案する新しいfail-safe 機構である多段階保護機構について述べる． 4 章では多段階保護機構を実現するために使われる保護マネージャについて説明し，そこで使われている保護の実装技術について述べる． 5 章では多段階保護機構の有効性を調べるために行った実験について述べる． 6 章では多段階保護機構の汎用性についての議論を行う．そして最後に7 章で本稿をまとめる．

2. Fail-safe 機構

OS の機能を容易に拡張できるようにする拡張可能OS がさかんに研究されている．この章では， OS を拡張するためのプログラムが意図しない動作をしたときでも， OS を守ることができるようにするfail-safe 機構について述べる．そして，これまでの拡張可能OS でどのようにfail-safe 機構が実現されており，そこにどのような問題があるのかについて論じる．

2.1 Fail-safe 機構の必要性

従来のOS では，ユーザがOS の機能を拡張するのは難しかった．たとえば，ソースコードにパッチを当ててOS 自体を再コンパイルしたり， OS にバイナリパッチを当てたりしなければならない． OS の再コンパイルにはOS のソースコードが必要になるが，商用のOS のソースコードはかなり高価なものであり，一般のユーザが手に入れるのは難しい．またバイナリパッチを当てる場合でも，パッチを当てる順番や組合せが問題になることがある．いずれにせよ， OS を拡張する作業をするときにはシステム全体を停止させなければならない．

この問題を解決するために， OS を拡張可能にするという研究が近年さかんに行われている．拡張可能OS では，拡張モジュールと呼ばれるプログラムを動的にOS に追加することによって，システムを停止させることなく， OS の機能を拡張することができる．たとえば，データベースを提供するベンダは，その性能を良くするために，そのデータベースに特化したファイルシステムも同時に提供したいということが考えられる．拡張可能OS は，ユーザにほとんど負担をかけずに，そのファイルシステムをOS に組み込むことを可能にする．

しかしながら拡張可能OS では，拡張モジュールのエラーからOS を守るためにfail-safe 機構が必要になる． OS ベンダが提供するOS カーネルと違って，サードパーティベンダも作成する拡張モジュールにエラーフリーであることを要求するのは現実的ではない．拡張しやすいOS であるためには，多少不安定な拡張モジュールが組み込まれても， OS 全体の安定性が損なわれない頑強さを持っている必要がある．そのような頑強さを持っていれば，拡張モジュールの開発作業は楽になり，また，いくつかの拡張モジュールを組み込んだ結果，OS が不安定になった場合でも，その原因となった拡張モジュールの特定に手間どるような事態を避けられる．

そのためにfail-safe 機構は，拡張モジュールのエラーを検出し，そのエラーから回復する機能を提供すべきである．拡張モジュールのエラーは， OS に悪影響を与える前に，できるだけ早い段階で検出されなければならな

い．さらに，エラーの原因を特定できるように，できるだけ正確に検出されることも重要である．そしてエラーが検出されたときには，OS をその影響から守るために，エラーからの回復を行わなければならない．いつエラーが検出されても回復できるように，fail-safe 機構はログをとるなど，つねに回復のために準備しておく必要がある．

2.2 これまでのシステム

fail-safe 機構を達成するために，これまでに様々な技法が提案されてきた．しかし，強力な fail-safe 機構ほど大きなオーバヘッドをともなうため，性能が重要な場合，しばしば fail-safe の機能をあきらめなければならなかった．

NetBSD などいくつかの UNIX システムでは，Loadable Kernel Module（LKM）と呼ばれる拡張モジュールを OS に追加することを可能にしている．LKM はあたかもカーネルの一部であるかのように実装され，実行時にカーネルと動的にリンクされる．リンクされた後は最初からカーネルに作り込まれたものと同様に振る舞うので，性能を非常に良くできる．その代わり，fail-safe 機構についてはまったく考えられていないので，拡張モジュールのエラーは OS 全体に影響を与える．これを回避するために，LKM に fail-safe 機構を導入する研究も進められている[10]．

Mach[1] のようなマイクロカーネル技術を用いた OS は，拡張可能 OS として利用することができる．マイクロカーネル技術を用いた OS では，拡張モジュールはユーザプロセスとして実装され，Inter Process Communication（IPC）を使ってカーネルや OS サーバと通信を行う．それゆえ，拡張モジュールのエラーが OS の他の部分に影響を及ぼすことはなく，エラーの検出された拡張モジュールはカーネルによって安全に終了させられる．よって十分な fail-safe 機構が提供されているといえるが，拡張モジュールと OS の他の部分の間の IPC や，コンテクストスイッチのオーバヘッドが大きく，実行性能が大幅に犠牲になっている．

拡張可能 OS である SPIN[2] では，言語の支援によって fail-safe 機構を達成している．拡張モジュールはカーネルにダウンロードされ，メモリアクセス違反を起こさないように，型安全な言語である Modula-3[4] で記述される．メモリアクセス違反はエラーの一部にすぎず，比較的対処しやすいのだが，対処が難しいその他のエラーについては考えられていない．また，OS を記述するのに一般に使われている C（C++）言語の代わりに型安全な言語を使うので，プログラミングを制限する可能性がある．

VINO[7] は SPIN と同様に，拡張モジュールをカーネルにダウンロードするが，メモリアクセス違反を検出するために Software Fault Isolation（SFI）[8] を使う．さらに拡張モジュールが使用できる資源の量を制限したり，一定期間が過ぎたら自動的に資源を解放することでデッドロックの問題を解決している．またエラーから回復するために，カーネル・トランザクションを提供している．このように様々なエラーに対する fail-safe 機構を実現しているが，fail-safe の能力を変えることはできないので，つねに一定の負荷がシステムにかかる．

Chorus[5] はマイクロカーネル技術を用いた OS であるが，拡張モジュールの fail-safe 機構の有無を選択することができる．拡張モジュールはまず，十分な fail-safe 機構を実現することができるユーザプロセスとして作られる．そしてデバッグが完全に終わったら，fail-safe 機構のオーバヘッドを受けないカーネル空間にロードすることができる．この考えは我々のものに近いが，Chorus の場合，拡張モジュールのデバッグが終わって完全にバグがなくなるまでは，完全な fail-safe 機構のオーバヘッドを受け続けてしまう．

3. 多段階保護機構

拡張可能 OS には fail-safe 機構が必要であるが，そのオーバヘッドが問題になる．そこで我々は，そのオーバヘッドを回避することができる新しい fail-safe 機構を提案する．

3.1 多段階保護機構

我々は，完全な fail-safe 機構はつねに必要なわけではないと考える．それは，信頼できるベンダの提供する拡張モジュールなどのように，拡張モジュールに悪意がないと仮定してもかまわない場合が多くあると考えられるからである．しかし，SPIN や VINO など多くの拡張可能 OS のように，あらゆるユーザが拡張モジュールを OS に組み込める場合には，一部のユーザが自分で作った悪意ある拡張モジュールを組み込んだり，信頼できない拡張モジュールを組み込んでしまうことがありうる．こうした場合に備えようとすると，つねに完全な fail-safe 機構が必要になる．それに対して，システムの管理者だけが拡張モジュールを OS に組み込めるようにすれば，悪意のある拡張モジュールが組み込まれる可能性は非常に低くなる．このような状況では，拡張モジュールをできるだけ効率良く実行しながら，そのエラーを検出して，OS を守ることが重要になると考える．この場合，完全な fail-safe 機構でなくても有用である．

さらに，拡張モジュールのバグは次第に減るので，

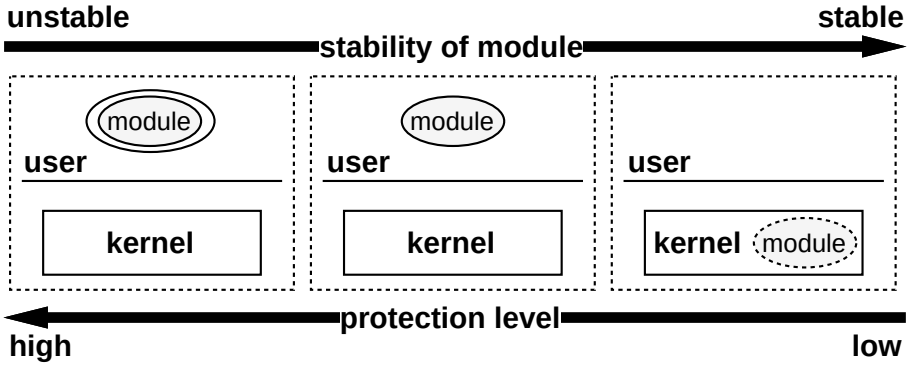


図 1 拡張モジュールの安定度に応じた保護レベルの選択
Fig. 1 Choice of a protection level corresponding to the stability of an extension module.

我々はそれに合わせて fail-safe 機構の保護レベルを段階的に変えられるべきであると考える．たとえば，拡張モジュールの開発時には多くのバグがあるので十分な fail-safe 機構が必要だが，リリース時にはほとんどなくなるので fail-safe 機構は必要ない．また発生するエラーの種類も，最初は不正メモリアクセスが頻繁に起こるのでメモリ保護が必要だが，次第にそのようなエラーは減り，デッドロックなどタイミングに依存したエラーだけに対処すればよくなる．

以上の考察に基づき，新しい fail-safe 機構である多段階保護機構を提案する．この機構は拡張モジュールを変更なしに，様々な保護レベルで OS に組み込むことを可能にする．この機能により，ユーザは拡張モジュールの安定度に応じてその保護レベルを変更し，不要な fail-safe 機構によるシステムの性能低下を回避することができる．拡張モジュールが安定していれば簡略化した fail-safe 機構を用いて性能向上をはかり，不安定ならば性能を犠牲にして完全な fail-safe 機構を用いた方がよい（**図 1**）．

多段階保護機構は拡張モジュールの保護レベルを変更するために，エラーの検出や回復の能力を変える．保護レベルを低くするには，一部のエラーを検出しないようにしたり，一部のエラーを回復しないようにする．たとえば，検出のオーバヘッドを減らすために，データの不正な読み出しを検出しないようできる．また，回復のためにとるログのオーバヘッドを減らすために，ログをとらないようにもできる．

多段階保護機構は拡張モジュールの保護レベルを容易に変更できるようにするために，拡張モジュールが従うべき API を提供する．この API が保護の実装の違いを隠蔽するので，拡張モジュールがこの API に従う限りは，保護レベルの変更の際にソースコードを変更する必要はない．

3.2 応 用 例

多段階保護機構がどのように使われるかを，拡張モジュールの開発者とユーザのそれぞれの観点から述べる．

3.2.1 拡張モジュールの開発

1 つの保護レベルだけでは，拡張モジュールの開発を容易にするのには不十分である．モジュールの安定度によって，エラーの種類や頻度は変わってくるので，開発を進めていく間に保護レベルを変えられるようにするべきである．

デバッグ時には，たとえ著しい性能低下を被るとしても，多段階保護機構は拡張モジュールに対して十分な保護をかけるべきである．それによって，拡張モジュールのエラーは可能な限り早く検出され，プログラマは正確なエラー情報を得ることができるからである．正確なエラー情報を得ることができれば，プログラマはより簡単にエラーの原因を特定して修正できるようになる．さらにエラーが検出された後には，その拡張モジュールを安全に OS から切り離すことができる．

ベータテスト時には，多段階保護機構は拡張モジュールに十分な保護をかける必要はない．むしろテストユーザがその性能に満足するように，できるだけ速く動かせるようにするべきである．なぜなら拡張モジュールの性能が良ければ，より多くのテストユーザに使ってもらえ，より多くのエラーを発見することができるからである．この段階では拡張モジュールはかなり安定していると考えられるので，デッドロックのようなタイミングに依存するエラーだけを検出すればよい．

そしてリリース時には，拡張モジュールをカーネルの中に組み込むことができる．このときの拡張モジュールにはまったく保護がかからないので，カーネルの中に手

で作り込んだものとほぼ同等の性能を得ることができる．

3.2.2 サードパーティ製の拡張モジュールの使用

不安定でしばらく使っていると異常終了するけれども，どうしても使いたいサードパーティ製の拡張モジュールも存在する．たとえば，OS ベンダが公式にはサポートしていない，CD-ROM チェンジャなどのデバイスドライバや，PC UNIX における NTFS などのファイルシステムが含まれる．このような拡張モジュールでも，短時間なら正常に機能するので，ユーザがその機能を必要としているなら許容することができる．しかし，長時間使った拡張モジュールが異常終了するたびに OS を巻き込まないようにして，OS は安定に保たれるべきである．

多段階保護機構を使えば，このような不安定な拡張モジュールを安全に動かすことができる．そして拡張モジュールが異常終了したときには OS から安全に切り離され，OS は守られる．しかしながら，多くのサードパーティ製の拡張モジュールは安定しているので，fail-safe 機構は必要とされない．多段階保護機構はこのような拡張モジュールを保護なしで動かすことを可能にし，性能低下を防ぐことができる．

4. ファイルシステムのための多段階保護機構

我々は，ファイルシステムに対して多段階保護機構を実現するシステムを NetBSD 1.2 上に実装した．ここではまず，このシステムの概要を説明し，どのように拡張モジュールの保護レベルを変更することができるかについて述べる．次に保護レベルの変更で重要な役割を果たす保護マネージャについて説明し，その中で使われる保護の実装技術について述べる．

4.1 システム概要

本システムでは，OS の機能拡張を拡張モジュールと呼ばれる，カーネルから独立したプログラムとして記述する．この拡張モジュールがカーネルの提供する機能を利用したり，カーネルのデータを操作するときには，保護マネージャと呼ばれる，OS が提供するライブラリを介して行う．この保護マネージャはカーネルのデータ構造に合わせて作られており，拡張モジュールが安全にカーネルデータを操作できるように fail-safe 機構を実現している．本システムでは，この保護マネージャを交換することによって，拡張モジュールの保護レベルを変更することができる．拡張モジュールと保護マネージャ，カーネルの関係は保護レベルにより変化するが，通常，拡張モジュールはユーザプロセスとなり，保護マネージャはそのプロセスとリンクされるライブラリとなる．

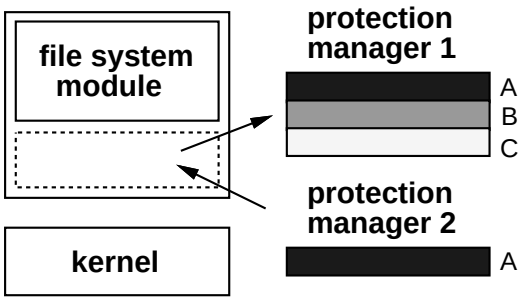


図 2 3 種類の保護（A,B,C）を行う保護マネージャ 1 を 1 種類の保護（A）しか行わない保護マネージャ 2 に交換する様子
Fig. 2 Exchange of protection manager 1 (protection A, B, C) with protection manager 2 (protection A).

ただし，保護レベルが十分に低くなると，拡張モジュールと保護マネージャはカーネルとリンクされ，カーネル内に組み込まれる．

4.2 保護レベルの変更

本システムでは，検出できるエラーや回復できるエラーの種類が異なっている保護マネージャが複数用意されており，ユーザはこれらの中から適切なものを選ぶことによって，拡張モジュールの保護レベルを変えることができる．拡張モジュールの保護レベルを変更する際には，ユーザは望む保護レベルを提供する保護マネージャを拡張モジュールに再リンクすればよい．ただし，保護マネージャが性能向上その他の目的のためにマクロを使っている場合には，再コンパイルも必要になる．その後，保護レベルを変えた拡張モジュールを有効にするために，拡張モジュールを起ち上げ直す．現在の実装では，保護レベルが十分に低くなり，拡張モジュールを配置するアドレス空間がユーザ空間からカーネル空間に変わった場合に限り，計算機の再起動も必要になる．保護マネージャを交換して保護レベルを低くする場合の概念図を**図 2** に示す．

4.3 保護マネージャ

4.1 節と 4.2 節で述べたように，保護マネージャは多段階保護機構を実現するために重要な役割を果たす．ここでは，まず保護マネージャの構成について述べ，次に保護マネージャの提供する API と，対象としているエラーについて説明する．

4.3.1 構　　成

保護マネージャは，大きく，拡張モジュールを管理する部分，カーネルからのアップコールを処理する部分，安全にカーネルデータを操作する部分に分かれる．拡張モジュールを管理する部分では，拡張モジュールをカーネルに登録したり，カーネルから削除したりする．特に，拡張モジュールが異常終了した場合には，責任を

表 1 OS の提供する専用システムコール
Table 1 Private system calls provided by our operating system.

システムコール	機能
modregist/modunregist	モジュールの登録，削除
shmprotect	共有メモリの保護
lock_and_wait	資源のロックなどの通知
logctl	ログの制御

持ってカーネルから切り離す．アップコールを処理する部分では，シグナルとして実装されているカーネルからのアップコールを受け取ったときに，引数の検査などの必要な処理をしてから，拡張モジュールを駆動する．安全にカーネルデータを操作する部分では，拡張モジュールのエラーによってカーネルデータが破壊されないように，カーネルデータの置かれているメモリ領域の保護，カーネルデータの複製とその一貫性保持，カーネルデータの検査などを行う．我々の OS は，このような保護マネージャを実現するために，**表 1** のようなシステムコールを提供している．

4.3.2 API

保護レベルの変更の際に拡張モジュールのソースコードを書き換えずに済ませるために，保護マネージャは拡張モジュールが従うべき API を提供する．この API は拡張モジュールがカーネルデータを操作するための API や，カーネルからのアップコール時に呼ばれるコールバック関数を登録するための API からなる．そのほかにも，デッドロックを検出できるように，資源のロックやアンロックを行う API なども存在する．すべての拡張モジュールはこの API に従わなければならないので，拡張モジュールに対する制限になりうるが，API に従うことによるソースコードの書き換え不要というメリットは，それを上回ると考えられる．当然，拡張モジュールに悪意がある場合はこの API に従うことを期待できないが，現実的には，組み込まれる拡張モジュールには悪意がないと仮定してよい場合がほとんどであると考えられる．

4.3.2.1 カーネルデータの操作

保護マネージャは，拡張モジュールが vnode や buf のようなカーネルデータを操作するための API を提供している．このような API が必要になるのは，拡張モジュールが直接カーネルデータを操作することができないからである．拡張モジュールがユーザプロセスである場合，保護マネージャはカーネルとの間に共有メモリを張って通信を行い，操作対象となるカーネルデータはこの共有メモリ上に存在する．しかし，この共有メモリは保護マネージャによって保護されるので，拡張モジュールが直接アクセスすることはできない．そこで保護マネージャによって提供される API を使って間接的にカーネルデータを操作する．**表 2** にこの API の一例を示す．さらにこの API は，拡張モジュールに抽象度の高いカーネルのデータ構造を見せるために，カーネルのデータ構造の一部を隠蔽する．これにより，エラーの原因になりやすいポインタ操作や複雑な操作は隠蔽され，拡張モジュールの開発者の負担も減る．

表 2 カーネルデータを操作する API の例
Table 2 Examples of API to manipulate kernel data.

API	機能
Bread	ファイルから buf に読み込む
Vref	vnode の参照回数を増やす
GetNextBlk	buf 鎖の次の要素を取り出す
MCbuild	mbuf 鎖にデータを詰める

表 3 アップコールの例
Table 3 Examples of upcall.

アップコール	要求
VOP_READ	ファイルを読む
VOP_LOCK	ファイルをロックする

また，保護マネージャは，カーネル内で提供されている API と同等の機能を実現する API も提供している．拡張モジュールがユーザプロセスである場合は，カーネル内で提供されている API を直接使うことができないからである．たとえば，時刻を得る関数や，NFS[6] サーバが UFS などカーネル内にあるファイルシステムを呼び出すための関数などがあげられる．これらの機能を実現するために，保護マネージャは内部でシステムコールを呼んだり，エミュレートしたりする．

4.3.2.2 コールバック関数の登録

保護マネージャは，カーネルからのアップコールによって拡張モジュールを駆動できるように，コールバック関数を登録する API を提供している．カーネルからのアップコールは，ファイルシステムの Virtual File System（VFS）[3] 層から，まず，保護マネージャに対して発行される．そして保護マネージャは，アップコールの引数として渡されるデータ構造を，拡張モジュールが使用する抽象度の高いデータ構造に変換し，その内容を検査してから，対応するコールバック関数を呼び出す．アップコールの例を**表 3** に，カーネルからのアップコールを処理する様子を**図 3** に示す．

4.3.3 対象とするエラー

保護マネージャは，不正メモリアクセスを検出・回復するエラーの対象としている．不正メモリアクセスには 2 種類あり，1 つは例外が発生するメモリアクセス，もう 1 つは意味的に不正なデータの変更である．例外が発

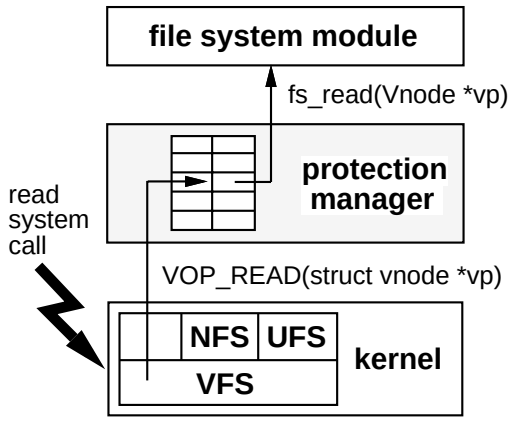


図 3 カーネルからのアップコールによって保護マネージャが対応するコールバック関数を呼び出す様子
Fig. 3 A callback function called by an upcall from the kernel.

生するメモリアクセスは，非常に容易に検出することができる．たとえば，ユーザプロセスがカーネル空間にアクセスするなど，他のアドレス空間のメモリを不正に読み書きした場合には，セグメンテーション違反が発生する．また，不正なアラインメントでメモリをアクセスすると，アラインメント違反が発生する．

一方，意味的に不正なデータの変更に関しては，不正な変更によってそのデータが意味的にとりうる範囲を超えたものをエラーとして検出することができる．たとえば，データの参照回数は負になることはないので，不正な変更によって負になった場合には，エラーとして検出される．また，データが存在するメモリ領域が決まっている場合，その領域外を参照するポインタはエラーとして検出される．

また，保護マネージャは，デッドロックも検出・回復するエラーの対象としている．たとえば，あるファイルシステムの中で 2 つのスレッドが動いていて，それぞれ別個のファイルをロックしているときに，互いに相手がロックしているファイルをロックしようとするとデッドロックに陥る．

4.4 保護の実装技術

保護マネージャは不正メモリアクセスとデッドロックを検出・回復するために，いくつかの実装技術を用いる．この実装技術の組合せ方を変えることで，様々な保護レベルの fail-safe 機構を実現することができる．ただし，すべての組合せが可能なわけではない．

4.4.1 不正メモリアクセスの検出

4.4.1.1 例外が発生するメモリアクセス

保護マネージャは不正メモリアクセスを検出するために，アドレス空間切替えを利用する．拡張モジュールをユーザプロセスとして作成し，カーネルと異なるアドレス空間に配置すれば，拡張モジュールによるカーネルメモリへの不正なアクセスを禁止できる．このように不正メモリアクセスによって例外が発生する場合には，検出は非常に容易である．しかしこの保護のオーバヘッドは，コンテクストスイッチやアドレス空間の間でのデータコピーなどのために，かなり大きくなる．そこでこのオーバヘッドを減らすために，拡張モジュールをカーネル空間に配置することも可能になっている．

拡張モジュールをユーザプロセスとして作成し，カーネルと異なるアドレス空間に配置する場合，カーネルと相互に通信するために共有メモリが必要になる．しかしこのような共有メモリはアドレス空間の分離によるカーネルの保護を破る．そこで保護マネージャのコードが実行されているとき以外は，拡張モジュールによる共有メモリの不正メモリアクセスを防ぐために， mmap や mprotect のようなシステムコールを使って，仮想記憶機構によるメモリ保護を行う．このようなシステムコールだけでは，ユーザプロセスとして作成された拡張モジュールのスタックやヒープへの不正アクセスを防ぐことはできない．しかし，そのようなエラーは拡張モジュール内部のエラーであり，カーネルデータを直接破壊することはないので，本システムでは対処しない．

本システムでは，共有メモリに対する保護の方法を選択することで，保護レベルを変えることができる．共有メモリ上のデータの破壊を防ぎ，不正な読み出しも検出したい場合には，拡張モジュールからは直接アクセスできないように，共有メモリをアンマップすればよい．一方，データの破壊だけを防ぎたいなら，共有メモリを読み出し専用にするだけでよい．この場合は，不正な読み出しの検出はできなくなり，エラーを早く正確に検出できなくなるかもしれないが，完全にアンマップするのに対してオーバヘッドを低くおさえることができる．さらに共有メモリに対する不正メモリアクセスを検出する必要がないなら，保護をまったくしないこともできる．この場合は保護のオーバヘッドを完全になくすことができる．

共有メモリに対する保護は，拡張モジュールと同一アドレス空間にある保護マネージャが行っているので，拡張モジュールに悪意があれば，共有メモリ上のデータを破壊することも可能である．しかし 3.1 節で述べたように，多くの場合，拡張モジュールには悪意はないと考えられるので，このような不完全な保護でも有用である．

4.4.1.2 意味的に不正なデータの変更

保護マネージャは，意味的に不正なデータの変更を検出するために，カーネルデータを複製し，その内容を検

査する．拡張モジュールは複製されたカーネルデータを自由にアクセスし，必要に応じて保護マネージャがそれをカーネルに書き戻す．そのときに保護マネージャは，データ構造についての様々な知識を利用して，そのデータを検査をする．たとえば，データの参照回数は 0 以上である，バッファの大きさの最小値と最大値は決まっている，あるデータのポインタは必ず共有メモリ上のアドレスを指している，などである．このような検査が可能なのは，保護マネージャが，複製するカーネルのデータ構造（vnode や buf など）に合わせて作成されているからである．

本システムでは，どの種のデータをどの程度検査するかを選択することで，複製による保護のレベルを変えることができる．たとえば，ポインタを手繰って調べるのをやめることで，ポインタによるループの形成を調べる必要がなくなり，オーバヘッドを減らすことができる．その代わりエラーの検出が遅れたり，できなくなったりする可能性がでてくる．

4.4.2 不正メモリアクセスからの回復

不正メモリアクセスのような続行不可能なエラーが検出された場合，もし拡張モジュールがカーネルデータを変更していたならば，OS を不安定にしないために，回復時に元に戻さなければならない．カーネルデータを元に戻せるようにするために，保護マネージャはログを用意し，拡張モジュールによるカーネルデータを変更する操作を記録する．そして回復時には，ログを検査して，その中に記録されている操作を元に戻す操作を実行する．たとえば，ファイルをロックする操作が記録されていれば，そのロックを解放する操作を行う．

本システムでは，どの操作をログに記録するか，どの操作を元に戻すかを変えることで，ログへの記録と回復のオーバヘッドを調節することができる．回復時に，拡張モジュールがカーネルに与えた影響を完全に除去できるようにするためには，すべての操作をログに記録し，すべての操作を元に戻す必要がある．その代わりに，一部の操作だけをログに記録するようにすれば，ログへの記録と回復のオーバヘッドを減らすことができる．たとえば多少のメモリリークは元に戻さなくても，カーネルにたいした影響を残さないので，無視できることが多い．

4.4.3 デッドロックの検出

本システムでは，デッドロックを検出するために，定期的にデッドロック状態になっていないかを検査する．拡張モジュールは，vnode などの資源をロックするとき，ロックを解放するとき，ロックを待つときに本システムに通知し，本システムではそれを記録する．デッドロックを検査するルーチンでは，その情報を基に wait-for-graph（WFG）を作成し，閉路ができていないかどうかを調べる．

本システムではデッドロックを検査する間隔を変えることで，デッドロックからの資源の保護のレベルを変えることができる．デッドロックを早く検出して，拡張モジュールのサービスが停止する時間を短くしたければ，この間隔を短くすればよい．この場合には，システムの性能低下は避けられない．逆にシステムの負荷を減らしたければ，この間隔を長くすればよい．ただし，デッドロックが生じてから検出するまでに時間がかかるようになり，拡張モジュールのサービスが停止する時間が長くなる．

4.4.4 デッドロックからの回復

デッドロックはタイミングに依存して起こるので，うまくデッドロックを解消すれば続行することができる場合が多い．そこでデッドロックが検出された場合には，デッドロック状態で停止している拡張モジュールが動けるように，デッドロックを解消する．そのために，本システムでは WFG の閉路を破壊するが，まずどこに閉路ができているのかを調べる．そして閉路中のロックのいずれかを一時的に解放することで，その閉路を破壊する．このときにデッドロックの本当の原因であるロックを解放すべきであるが，それを調べるのは難しいので，現在の実装ではランダムにロックを解放している．また，ロックを強制的に解放された拡張モジュールは，再びそのロックが取得できるようになるまでは，実行を停止される．

5. 実　　験

我々は多段階保護機構の有効性を調べるために，いくつかの実験を行った．この実験の目的は第 1 に，保護レベルを変更して fail-safe の機能を減らしたときに，実行性能が改善されることを調べることである．第 2 に，保護レベルが最大のときに，どの程度のオーバヘッドを被るかを調べることである．第 3 に，保護レベルを最小にしたときに，保護マネージャの API を共通にすることによるオーバヘッドがどの程度であるかを調べることである．

本システムで提供している保護の実装技術の組合せはかなり多いが，すべてについて実験を行うのは困難なので，その中から**表 4** のような特徴的な 5 種類の組合せを選び，実験を行った．これら 5 種類の組合せは，それぞれ**表 5** のようなメモリに関するエラーを検出することができる．この中で最も保護レベルの高いものを第 1 段階とし，最も低いものを第 5 段階とする．

表 4 特徴的な 5 種類の組合せ （* 共有メモリをアンマップする，** 共有メモリを読み出し専用にする）
Table 4 5 characteristic combinations (*Unmap shared memory, **Make shared memory read-only).

実装技術	1	2	3	4	5
共有メモリ保護	√*	√**			
複製	√	√	√		
アドレス空間切替え	√	√	√	√	
ログの記録	√	√	√	√	
デッドロックの検査	√	√	√	√	

表 5 各段階で検出可能なエラー
Table 5 Detectable errors at each protection level.

エラー	1	2	3	4	5
不正な共有メモリの読み出し	√				
不正な共有メモリの書き込み	√	√			
意味的に不正なデータの更新	√	√	√		
カーネルデータの不正アクセス	√	√	√	√	

我々は実験を行うにあたって，2 つのファイルシステム・モジュールを作成した．1 つは SMFS （Simple Memory File System）と呼ばれる簡単な RAM ディスクであり，もう 1 つは SNFS （Simple Network File System）と呼ばれる簡単な NFS である．

実験環境として，SMFS と SNFS のクライアントには SPARCstation 5 （MicroSPARC2/85MHz）を用い，SNFS のサーバには PC （Cyrix 6x86/133MHz）を用いた．ともに OS は多段階保護機構を追加した NetBSD 1.2 である．SNFS のサーバとクライアントは 10Mbps のネットワークで接続される．

5.1 保護のオーバヘッドの測定

まず，本システムで fail-safe 機構を実現するために使っている保護のオーバヘッドがどの程度であるのかを調べるために，ファイルシステムに対する基本的な操作であるファイルのコピーにかかる時間を測定した．コピーするファイルのサイズは 64KB で，1 回のシステムコールで扱うブロックサイズは 8KB である．この実験結果を**図 4** と**図 5** に示す．

この結果から保護レベルを低くすれば性能が良くなることが分かる．保護レベルの最も高い第 1 段階では最も低い第 5 段階に比べて，SMFS で 12 倍，SNFS で 2.2 倍のオーバヘッドを被っており，性能はかなり悪くなっている．それでもデバッグなどの目的のためならば実用に耐えないほどではない．

次に，ファイルシステムに対するより現実的な操作において，保護のオーバヘッドがどの程度であるのかを調べるために，NetBSD 1.2 の ps プログラムをコンパイルするのにかかる時間を測定した．このプログラムは，5 つのソースファイルと 2 つのヘッダファイルから

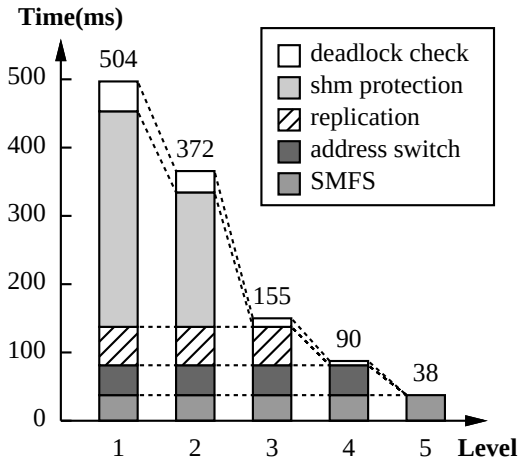


図 4 SMFS で 64KB のファイルのコピーに要する時間とその内訳
Fig. 4 The time needed to copy a 64KB file in SMFS and the breakdown.

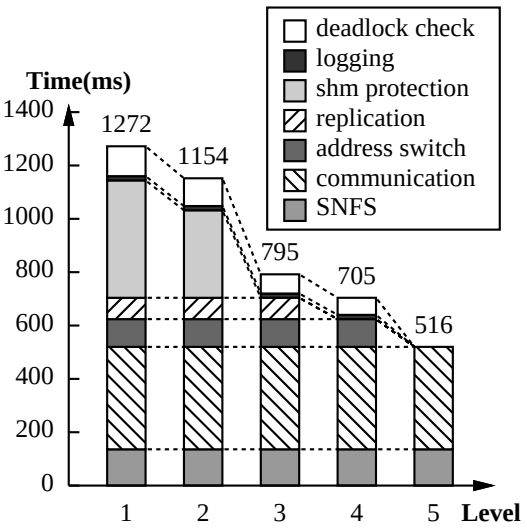


図 5 SNFS で 64KB のファイルのコピーに要する時間とその内訳
Fig. 5 The time needed to copy a 64KB file in SNFS and the breakdown.

なり，全体で約 2,000 行である．この実験結果を**図 6** と**図 7** に示す．この実験のような，より現実に近い状況では，SMFS の第 1 段階でも第 5 段階の 1.7 倍程度のオーバヘッドで済むので，十分実用に耐える．

5.2 API のオーバヘッドの測定

API を共通にすることでどの程度のオーバヘッドになるのかを調べるために，カーネルに作り込んで手で最適化したものと，保護レベルを最小にしてカーネルに組み込んだもの（第 5 段階）とを比較した．それぞれについて，64KB のファイルのコピー（ブロックサイズは

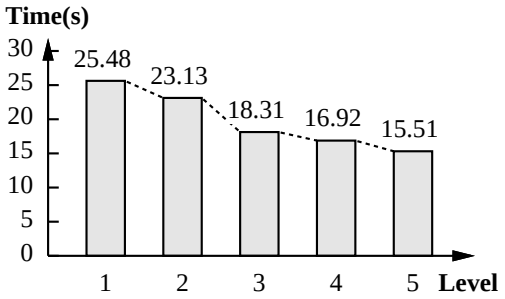


図 6 SMFS で ps のコンパイルに要する時間
Fig. 6 The time needed to compile ps in SMFS.

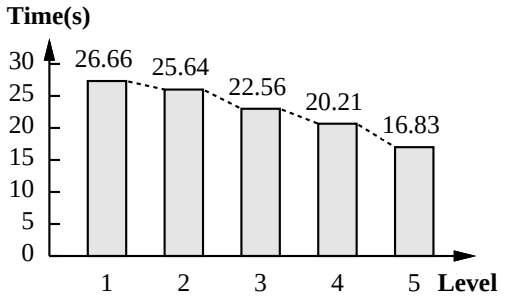


図 7 SNFS で ps のコンパイルに要する時間
Fig. 7 The time needed to compile ps in SNFS.

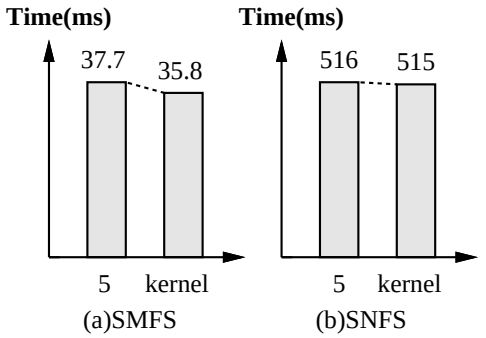


図 8 API のオーバヘッドの測定
Fig. 8 Measurement of API overhead.

8KB）を行った結果を**図 8** に示す．

API を共通にするオーバヘッドは，SMFS で 5%，SNFS で 0.1% 程度である．このオーバヘッドの内訳は，API を共通にするための余計な関数呼び出し，データ構造の変換にともなうコピー，オブジェクトサイズの増加によるキャッシュミスの増加などであると考えられる．この結果より，多段階保護機構を使って作られた拡張モジュールは，保護レベルを最小にしてカーネルに組み込めば，カーネルに作り込んで手で最適化したものに十分近い性能が得られることが分かる．

6. 多段階保護機構の汎用性

現在，我々は新たなファイルシステムを作成する場合にのみ多段階保護機構を適用している．ファイルシステムは OS の中でもかなりうまく切り分けられているサブシステムなので，拡張モジュールとして作成し，多段階保護機構を適用するのは比較的容易であった．同様にネットワークシステムもうまく切り分けられているので，新たなネットワークプロトコルを拡張モジュールとして作成し，多段階保護機構を適用することは可能であると考える．ただし，ファイルシステムのキャッシュのポリシだけを置き換えるといった拡張を行えるようにするためには，その拡張モジュールが異常終了したときにデフォルトのものに切り換えられるようにするなどの工夫を行う必要がある．

それに対して，CPU スケジューラなどは切り分けにくく，非常にタイミングに依存しているので，拡張モジュールとして作成するのはそれほど容易ではない．たとえば，CPU スケジューラを置き換える拡張モジュールは，そのプロセスの優先度にかかわらず呼び出せるようにする必要がある．また，クリティカルセクションを実現するためには，割込みの禁止を考慮する必要がある．このようなサブシステムを拡張モジュールとして作成し，多段階保護機構を適用できるようにするためには，さらなる考察が必要であると思われる．しかしながら，これらの問題を解決できれば，4.4 節で述べている保護の実装技術などはそのまま使えると考えられる．

7. ま と め

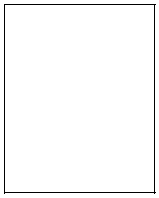
本稿では拡張モジュールを変更なしに，様々な保護レベルで OS に組み込むことができる多段階保護機構を提案した．この機構では保護マネージャを交換することにより，安定度に応じて拡張モジュールの保護レベルを変えることができる．我々は，ファイルシステムに対して多段階保護機構を実現するシステムを NetBSD 1.2 上に実装し，その有効性を確かめる実験を行った．この実験により，fail-safe の機能を減らせば，その分実行性能を改善できることが分かった．さらに保護レベルを最大にしたときでも，オーバヘッドは実用に耐える程度であり，保護レベルを最小にすれば，カーネルに作り込んで手で最適化したファイルシステムに十分近い性能が得られることが分かった．よって多段階保護機構は，拡張モジュールにバグがあるときにはその使用を支援し，拡張モジュールにバグがなくなったときには非常によい性能を得られる fail-safe 機構であるといえる．

謝辞 研究に関して適切な助言をくださった益田研究室の方々，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

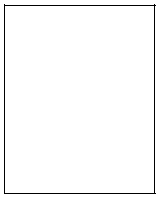
1) Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A. and Young, M.: Mach: A New Kernel Foundation for UNIX Development, *Proc. USENIX 1986 Summer Conference*, pp. 93–112 (1986).
2) Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, C. and Eggers, S.: Extensibility, Safety and Performance in the SPIN Operating System, *Proc. 15th ACM Symposium on Operating Systems Principles*, pp. 267–284 (1995).
3) McKusick, M. K.: The Virtual Filesystem Interface in 4.4BSD, *Computing Systems*, Vol. 8, No. 1, pp. 3–25 (1995).
4) Nelson, G.: *System Programming with Modula-3*, Prentice Hall (1991).
5) Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F., Kaiser, C., Langlois, S., Léonard, P. and Neuhauser, W.: CHORUS Distributed Operating System, *Computing Systems*, Vol. 1, No. 4, pp. 305–370 (1988).
6) Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D. and Lyon, B.: Design and Implementation of the Sun Network Filesystem, *Proc. USENIX 1985 Summer Conference*, pp. 119–130 (1985).
7) Seltzer, M. I., Endo, Y., Small, C. and Smith, K. A.: Dealing With Disaster: Surviving Misbehaved Kernel Extensions, *Proc. 2nd Symposium on Operating Systems Design and Implementation*, pp. 213–227 (1996).
8) 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).
9) 光来健一, 千葉滋, 益田隆司: 新規ファイルシステムの開発における OS の多段階保護機構の必要性, 情報処理学会研究報告 97-OS-76, pp. 37–42 (1997).
10) 多田好克, 中村嘉志, 林隆宏: カーネルの発展性と安全性に関する一考察, 情報処理学会研究報告 97-OS-76, pp. 61–65 (1997).

(平成10年3月26日受付)
(平成10年9月 7 日採録)



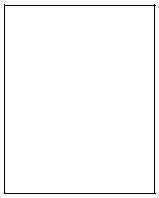
光来 健一

1975 年生． 1997 年東京大学理学部情報科学科卒業．現在，同大学大学院理学系研究科情報科学専攻修士課程在学中．オペレーティングシステムの拡張性に関する研究に従事．



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．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．



益田 隆司（正会員）

1939 年生． 1963 年東京大学工学部応用物理学科卒業． 1965 年同大学大学院修士課程修了．同年（株）日立製作所入社． 1977 年から筑波大学，1988 年 3 月から東京大学に勤務．現在，同大学大学院理学系研究科情報科学専攻教授．専門はオペレーティングシステム．