システムソフトウェアと
オペレーティング・システム 77−34
マルチメディア通信と分散処理 87−34
（1998. 2. 27）

拡張可能OSのfail-safe機構

光来 健一† 千葉 滋‡ 益田 隆司†

†東京大学大学院 理学系研究科 情報科学専攻
‡筑波大学 電子・情報工学系

要旨

fail-safe機構は拡張可能OSにとって重要である。しかし十分なfail-safeの機能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を避けられなかった。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。この機能により、ユーザは拡張モジュールの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。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、fail-safeの機能とファイルシステムの実行性能のトレードオフをとれることを確かめた。

Fail-safe mechanism for extensible operating systems

Kenichi Kourai† Shigeru Chiba‡ Takashi Masuda†

†Department of Information Science, Graduate School of Science, University of Tokyo
‡Institute of Information Science and Electronics, University of Tsukuba

Abstract

A fail-safe mechanism is significant for extensible operating systems to prevent erroneous extension modules from destroying the system. However this mechanism 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 embed an extensible module in the operating system at various protection levels without changing the source code, and thereby to avoid performance penalties by the excess capability for the fail-safety. We have implemented the multi-level protection for file systems of NetBSD 1.2 and showed that it enables the performance improvement of the file system if relaxing the fail-safety.

1 はじめに

必要に応じて新しいモジュールをOSに組み込むことができる拡張可能なOSが盛んに研究されている。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。しかし十分なfail-safeの機能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、実際に保護レベルを変更して、fail-safeの機能とファイルシステムの実行性能のトレードオフをとれることを確かめた。

2 多段階保護機構

拡張しやすい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。

2.1 fail-safe機構

拡張可能OSでは、ユーザがOSに新しいソフトウェア・モジュールを追加することによって、OSが提供するサービスを拡張することができる。例えば特定のアプリケーションに特化したファイルシステムを追加して、そのアプリケーションの実行性能を良くすることができる。しかし拡張可能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カーネルと違って、サードパーティなどが作成する拡張モジュールが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もっていれば、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取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時には、OSをその影響から守るために、エラーからの回復を行な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機構はログを取るなど、常に回復のために準備しておく必要がある。

この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をあきらめなければならなかった。例えばNetBSDなど一部のUNIXでは、loadable kernel module (lkm) を使ってシステムを拡張することができるが、性能を優先するために拡張モジュールには全く保護をかけず、完全にfail-safeをあきらめている。逆にMach [1] では、拡張モジュールをユーザプロセスとして実装するので、十分なfail-safe機構が実現できているが、その分、実行性能が犠牲になっている。

2.2 多段階保護機構

我々は「完全なfail-safe機構は常に必要なわけではない」という知見に基づき、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。

多段階保護機構は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。保護を弱めるには、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。例えば、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、回復のために取るログのオーバヘッドを減らすために、ログを取らないようにもできる。

多段階保護機構は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。

2.3 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。

2.3.1 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。

2.3.2 サードパーティ製の拡張モジュール

不安定で頻繁に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。例えば、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は頻繁に異常終了するかもしれないが、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、拡張モジュールが異常終了する度に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時には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。

3 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。本節では、このシステムがどのようなエラーに対してfail-safe機構を実現し、どのように保護レベルを変えることができるのかについて述べる。

3.1 対象とするエラー

ここでは、本システムがファイルシステム・モジュールに対するfail-safe機構を実現するために、具体的にどのようなエラーを対象にしているのかについて述べる。

3.1.1 不正メモリアクセス

本システムでは、不正メモリアクセスを検出・回復すべきエラーであると考えている。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。例えば、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取り得る範囲を越えた場合にだけ、エラーとして検出することができる。例えば、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、存在するメモリ領域が限定されているデータを指すポインタで、その領域外を参照しているものはエラーとして検出される。

不正メモリアクセスをしたファイルシステム・モジュールは続行不可能なので、終了させてOSから切り離す。ファイルシステムは一般のアプリケーションと違い、OSに密接に関わっているので、単純に終了させることはできない。そこでファイルシステム・モジュールがOSに与えた影響を除去して、OSが不安定にならないようにする。

3.1.2 デッドロック

本システムでは、デッドロックも検出・回復すべきエラーであると考えている。例えば、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時に、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。デッドロックを回避するために、デッドロックに陥いらないようなロックの取り方をさせることもできるが、デッドロックにならない場合でもロックが取れないことがあり、性能低下につながるので、本システムでは考えていない。

デッドロックはタイミングに依存しており、続行可能なエラーなので、デッドロックを解消することで回復を行なう。

3.2 API

本システムでは、ファイルシステムを書く際に従うべきAPIをライブラリによって提供する。このライブラリが提供するAPIは、拡張モジュールがカーネルデータを操作するインタフェースと、カーネルから拡張モジュールへのコールバックのインタフェースから成る。全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得るが、このAPIに従うことで、容易に保護レベルを変更できるという利益を得ることができる。

悪意のある拡張モジュールはこのAPIに従うことは期待できないが、我々はOSの拡張モジュールには悪意はないと考えている。なぜなら、OSの拡張モジュールはOSベンダやサードパーティなど、信頼できるベンダから提供されると考えられるからである。

3.2.1 カーネルデータの操作

本システムの提供するライブラリは、カーネルのデータ構造をより抽象度の高いデータ構造に変換し、それを拡張モジュールに見せる。vnodeやbufなどのカーネルのデータ構造のメンバの一部は、隠蔽され、直接アクセスを許されない。この隠蔽されたメンバにアクセスするには、ライブラリが提供す

API	機能
隠蔽されたメンバを操作する API	
GetNextBlk	buf 鎖の次の要素を取り出す
MCbuild	mbuf 鎖にデータを詰める
カーネル内で提供されている API	
Vref	vnode の参照カウンタを増やす
Bread	ファイルから buf に読み込む

表 1: カーネルデータを操作する API の例

種類	要求
VOP_READ	ファイルから読む
VOP_LOCK	ファイルをロックする

表 2: コールバックの例

る API を使用する。これにより、エラーの原因になり易いポインタ操作や複雑な操作はライブラリによって隠蔽され、ユーザが書く必要がなくなる。また、カーネル内で提供されている API を拡張モジュールでも利用できるように、それと同等の機能を実現する API をライブラリで提供する。これらの API の例を表 1に示す。

3.2.2 コールバック

カーネルから拡張モジュールを呼び出すコールバックは、直接行なわれるのではなく、ライブラリを介して行なわれる。コールバックを受け取ったライブラリは、あらかじめ登録されている情報を元に、拡張モジュールのコールバック関数を呼び出す。この際に、引数として渡されるデータ構造は、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される。コールバックの例を表 2に示す。

3.3 保護レベルの変更

保護レベルを変える際にユーザがすべきことは、ライブラリを交換することだけである。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ライブラリが複数用意されている。ユーザはこれらのライブラリの中から適切なものを選ぶことによって、保護レベルを変えることができる。ライブラリを交換しても同じ API を使うことができるので、ユーザは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。

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ライブラリを拡張モジュールに再リンクすればよい。ただし、ライブラリ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。その間、一時的に拡張モジュールによるサービスが中断してしまうが、わずかの間なのでファイルシステムの場合は問題ではない。現在の実装では、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。

3.4 実装技術

本システムで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。拡張モジュールの安定度に応じて、適切な実装技術を選ぶことによって、様々な保護レベルの fail-safe 機構を実現することができる。また、ユーザが検出したいエラーの種類に応じて、いくつかの実装技術を組合せて使用することができる。ただし、全ての組合せが可能なわけではなく、例えばアドレス空間を切替える際には、カーネルデータの複製は必須になる。

3.4.1 不正メモリアクセスの検出

本システムで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。拡張モジュールをカーネルと異なるアドレス空間に配置すれば、ハードウェアによって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、本システムでは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している。

拡張モジュールがカーネルと異なるアドレス空間に配置される場合、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ので、不正メモリアクセスから保護しなければならない。本システムでは共有メモリをハードウェアで保護することによって、可能な限り早く正確に不正メモリアクセスを検出する。

この共有メモリに対する保護を弱めるために、本システムでは保護の方法を変えることを可能にしている。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、共有メモリを完全にアンマップすればよい。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抑えることができる。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、保護を全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。

一方、意味的に不正なデータの変更を検出するために、本システムではカーネルデータを複製する。拡張モジュールはライブラリによって複製されたカーネルデータに自由にアクセスし、必要に応じてライブラリがそれをカーネルに書き戻す。その時にライブラリは、そのデータに対して可能な限りの検査をする。その際、データ構造についての様々な知識を利用する。例えば、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。その他に、複製とカーネルデータの対応がとれないデータもエラーとして検出される。

この複製による保護を弱めるために、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変えることを可能にしている。例えばポインタを手繰って調べるのをやめることで、ポインタがループを形成していないかどうかを気にしなくてよくなるので、その分、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、できなくなる可能性がでてくる。

3.4.2 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、本システムでは拡張モジュール毎にログを用意し、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。例えば資源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なう。

このログへの記録と回復のオーバヘッドを減らすために、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を可能にしている。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、全ての操作をログに記録し、全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。例えば多少のメモリリークは元に戻さなくても、カーネルに大した影響を残さないので、無視できることが多い。

3.4.3 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などのカーネルデータをロックする時、ロックを解放する時、ロックを待つ時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、その情報を元に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。

デッドロックからのカーネルデータの保護を弱めるために、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を可能にしている。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。

3.4.4 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。この時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行なっている。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。

4 実験

我々は、多段階保護機構を用いた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 [2]である。

これら2つのファイルシステムについて保護のオーバヘッドを測定する実験を行なった。この実験では、1回のシステムコールで読み書きするデータの大きさを8Kbyteとして、全体で64Kbyteのファイルのコピーを SMFS、SNFS について行なった。計算機には SPARCstation 5 (MicroSPARC2/85MHz, メモリ 32M) と、SNFS のサーバとして PC (Cyrix 6x86/133MHz, メモリ 64M) を用い、OS には多段階保護機構を組み込んだ NetBSD 1.2 を用いた。

この実験の結果を表 3に示す。アドレス空間切替えはコンテクスト・スイッチだけではなく、拡張モジュールへのアップコールや、拡張モジュールからのシステムコールのオーバヘッドも含まれている。デッドロックの検査全体でのオーバヘッドは、64Kbyteのファイルをコピーする間に検査する回数に依存する。この回数はデッドロックを検査する間隔と、コピーに要する時間によって決まる。なお比

保護	SMFS	SNFS
複製 (全データ)	59	82
アドレス空間切替え	44	104
共有メモリ保護 (アンマップ)	317	433
共有メモリ保護 (読み出し専用)	197	326
デッドロック検査 (1 回当たり)	0.1	0.1
ログの記録	0	21
fail-safe なしのコピー	38	516

表 3: SMFS と SNFS で 64Kbyte のファイルのコピーをする時の保護のオーバヘッドおよび fail-safe なしでコピーに要する時間 (ms)

較のために、fail-safe に関する処理を全く行なわない場合に、64Kbyte のファイルのコピーに要する時間も表に含めてある。

今回の実験では、1 回のデッドロックの検査のオーバヘッドは 100μs 程度であったが、大量のファイルを検索するプログラムのようにロックとその解放を繰り返すものでは 160μs 程度になった。拡張モジュールの数が増えて WFG が複雑になると、このオーバヘッドはさらに大きくなると考えられる。

ログを記録するオーバヘッドは、拡張モジュールがカーネルデータを変更する頻度に応じて変わる。頻繁に相互作用する場合は、このオーバヘッドは更に増すと考えられる。

また多段階保護機構で保護を最小にしたものと、最初からカーネルに作り込んだものとを比較すると、SMFS で 2.4%、SNFS で 0.1% 程度のオーバヘッドが測定された。保護レベルを変えられるようにする代償がこの程度のオーバヘッドなら、十分に有用であると考えられる。

5 関連研究

Mach [1] のようなマイクロカーネルでは、十分な fail-safe 機構が提供されている。拡張モジュールはユーザプロセスとして実装されるので、そのエラーが OS に影響を及ぼすことはない。エラーの検出されたモジュールはカーネルによって安全に終了させられる。しかし十分な fail-safe 機構を提供するために、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 [3] 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過ぎず、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。また型安全な言語を使うのは、プログラミングを制限する可能性がある。

VINO [5] 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し、メモリアクセス違反を検出するために SFI [6] 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。ただし、fail-safe 機構の能力を変えることはできないので、常に一定の負荷がシステムにかかる。

6 まとめと今後の課題

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。この機構により、ユーザは拡張モジュールの安定度に応じて、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、実際に保護レベルを変更して、fail-safe の機能とファイルシステムの実行性能のトレードオフをとれることを確かめた。

ファイルシステムだけでは汎用性に欠けるので、今後はネットワーク・システムなどの他のサブシステムについても、多段階保護機構を実装する予定である。また拡張モジュールを停止させずに保護レベルを変えられるようにして、よりシームレスにすべきであると考える。

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, *Proceedings of the USENIX 1986 Summer Conference*, pp. 93–112 (1986).

[2] Sandberg, R.: Design and Implementation of the Sun Network Filesystem, *Proceedings of the USENIX 1985 Summer Conference* (1985).

[3] Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, S. and Eggers, C.: Extensibility, Safety and Performance in the SPIN Operating System, in *Proc. 15th ACM Symposium on Operating Systems Principles*, pp. 267–284 (1995).

[4] Nelson, G.: *System Programming with Modula-3*, Innovative Technology, Prentice Hall (1991).

[5] Seltzer, M. I., Endo, Y., Small, C. and Smith, K. A.: Dealing With Disaster: Surviving Misbehaved Kernel Extensions, in *2nd Symposium on Operating Systems Design and Implementation*, pp. 213–227 (1996).

[6] Wahbe, R., Lucco, S., Anderson, T. E. and Graham, S. L.: Efficient Software-Based Fault Isolation, in *Proceedings of the 14th Symposium on Operating Systems Principles*, pp. 203–216 (1993).