Vol.2018-OS-143 No.14
2018/5/22

カーネル内部データのプロセス間分離による堅牢性の向上

杉本 学[1]　窪田 貴文[1]　河野 健二[1]

概要：コンピュータシステムの信頼性を損なう要因の一つに，オペレーティングシステムのカーネルフェイラがある．実際，Linux には 700 以上のフォールトが存在し，半年間に 187,000 件以上の障害レポートが報告されている．カーネルにおけるフェイラでは，エラーがカーネル全体に伝播する場合は少なく，多くはカーネル内のプロセスコンテキストに閉じたプロセスローカルエラーとなっている．そして，フェイラの約 73% はこのプロセスローカルエラーによるものである．本論文では，プロセスローカルエラーによるカーネルフェイラを検知しエラー状態を取り除くことで，カーネルの実行を継続する手法を提案する．プロセスローカルエラーでは，エラー状態がプロセスコンテキストに閉じているため，フェイラの発生したプロセスを強制終了することでカーネル内のエラー状態を回復させることができる．これにより，従来のカーネルではフェイラとなっていた場合でも，カーネルを停止させずに他のプロセスの実行を継続することができる．

キーワード：ソフトウェア障害，カーネル，エラー伝播，耐障害性

1. はじめに

コンピュータシステムには高い信頼性が求められる．一度システムが停止してしまうと，サービスの提供者や利用者に非常に大きな損害を与えてしまう．例えば，証券会社やクレジット会社の場合，1 時間に数百万ドル，インターネット通販会社では 1 時間に数十万ドルの損失となる [1]．また，システムの停止による復旧作業とその障害の対応に，運用コストの 30% から 50% という負担がかかる [2]．加えて，システムのダウンタイムは致命的であり，サービスの可用性を大きく低下させる．

コンピュータシステムの信頼性を損なう要因の一つに，オペレーティングシステム (OS) のカーネルフェイラがある．カーネルフェイラは，OS の上で動作する全てのアプリケーションの停止に繋がるため，致命的な障害となる．実際，Linux にはどのバージョンにも 約 700 ものフォールトが存在し [3]，半年間に 187,000 件以上の障害レポートが報告されている [4]．このことから，OS におけるフェイラの発生は避けられない状況にある．

一般的に，フェイラはエラー状態がシステム内を伝播していくことで発生する． このエラー伝播により OS がフェイラに陥った場合，エラーの発生源や伝播過程を特定することは難しい．そのため，エラー伝播によるカーネルフェイラからの回復は困難である．しかし，一般的にエラー状態がカーネル全体に伝播するケースは少なく，多くがカーネル内の単一のプロセスコンテキスト内でエラー伝播して発生する**プロセスローカルエラー**であることが知られている [5]．そしてフェイラの約 73% がこのプロセスローカルエラーによるものである．また，カーネルフェイラの約 90% はエラーが発生したサブシステム内で完結する [6]．これより，フェイラはカーネルの全体がエラー状態となって発生しているわけではなく，カーネルのごく一部のデータのみが壊れて発生している可能性が非常に高いと言える．

そこで，本論文ではプロセスローカルエラーによるフェイラを検知しエラー状態を取り除くことで，カーネルの実行を継続する手法を提案する．プロセスローカルエラーでは，全てのプロセスコンテキストで共有して使用されているデータにエラー伝播していないため，エラー状態のプロセスコンテキストのみを捨てることでカーネル全体のエラー状態を取り除くことができる．全てのプロセスコンテキストで共有されているデータにエラーが伝播することで発生する**カーネルグローバルエラー**と区別するため，カーネル内部のデータ領域をプロセスローカルデータとカーネルグローバルデータの二つに分離する．カーネルグローバルデータに書き込みをせずにフェイラとなった場合，エラー状態がカーネル内のプロセスコンテキストに閉じているため，フェイラを起こしたプロセスを強制終了させることでエラー状態を取り除く．これにより，従来のカーネル

1　慶應義塾大学
Keio University

© 2018 Information Processing Society of Japan

ではフェイラとなっていた場合でも，カーネルを停止させずに正常な状態で他のプロセスの実行を継続することができる．

提案手法の機構をマサチューセッツ工科大学が開発をしているリサーチカーネルである xv6 [7] に実装し，ソフトウェアフォールトインジェクション (SFI) による堅牢性の実験を行った．その結果，通常の xv6 ではカーネルフェイラを引き起こしたフォールトであっても本機構を導入した xv6 では，プロセスローカルエラーとして検知し，カーネルを停止させることなく，正常な状態で動作を継続させることが可能であることを示す．

本論文の構成を以下に示す．第 2 章では，フェイラに至る過程で発生するエラー伝播の詳細について述べる．第 3 章では，本研究の提案手法であるカーネルの設計について説明する．第 4 章では，提案手法の実装の詳細を説明する．第 5 章では，SFI による堅牢性を検証した実験について述べる．第 6 章では，本研究の関連研究について紹介する．第 7 章では，まとめについて述べる．

2. エラー伝播

一般的にフェイラはエラーが伝播することで発生する．エラー伝播とは，エラー状態にあるデータを用いて計算した結果，その結果もエラー状態となり，新たにエラーが発生し伝播していく一連の流れである．エラー状態にあるデータとは，フォールトの実行によって期待とは異なる状態になっているデータのことである．

2.1 エラー伝播の範囲

既存研究からエラーが伝播する範囲には特徴があることが知られている．ここでは，エラー伝播に関する既存研究を取り上げ，実際の Linux カーネルではどのようにエラーが伝播しカーネルフェイラに至るのかを説明する．

文献 [5] の研究ではエラー伝播の範囲をカーネルグローバルエラーとプロセスローカルエラーの 2 つに分類している．カーネルグローバルエラーは，カーネルの共有データにエラーが伝播して発生するエラーである．カーネルの共有データとは，全てのプロセスコンテキストが共有して使用しているデータであり，主にロックを使用して同期をしなければ正しく利用できないものである．プロセスローカルエラーは，カーネル内のプロセスコンテキストのデータにエラー伝播が閉じるエラーである．プロセスコンテキストのデータとは，プロセス毎のカーネルスタックなどのカーネル全体で共有していないデータである．

カーネルグローバルエラーとプロセスローカルエラーの概要について説明する．カーネルグローバルエラーを示したものが図 1 である．プロセス 1 のカーネルコンテキストでフォールトを実行しデータがエラー状態となったとする．このエラーはプロセス 1 のカーネルコンテキストからカーネルの共有データに伝播し，最終的にカーネルはフェイラに至る．カーネルの共有データにエラーが伝播しているため，他のプロセスコンテキストがエラー状態の共有データを参照した場合，そのプロセスコンテキストにもエラーが伝播する可能性がある．プロセスローカルエラーを示したものが図 2 である．このエラーの場合，カーネルのエラー状態はプロセス 1 のカーネルコンテキストのみであり，他プロセスのカーネルコンテキストやカーネルの共有データにエラー伝播はしていない．

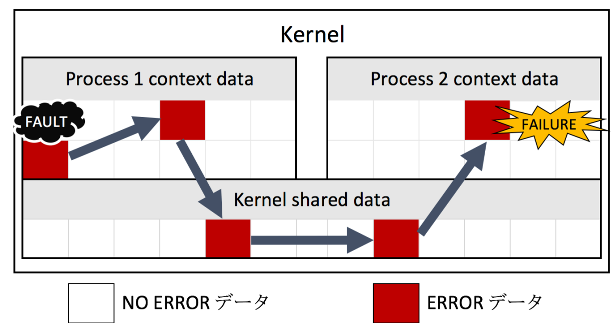

図 1　カーネルグローバルエラー

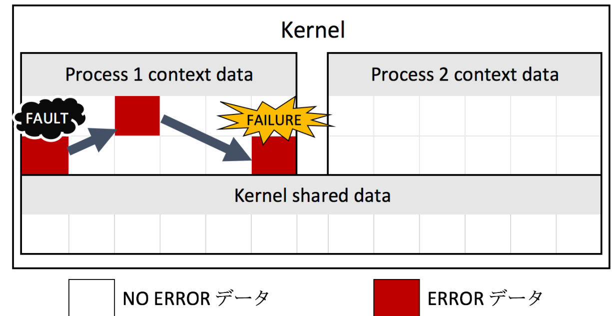

図 2　プロセスローカルエラー

OS ではカーネルグローバルエラーによるフェイラの発生は少なく，エラー状態はプロセスコンテキスト内で閉じることが多いことがわかっている．Linux のカーネルフェイラの約 73% はこのプロセスローカルエラーによるものである [5]．これは，エラーの多くがローカル変数や返り値などといった，関数内の変数により伝播していたためである．

また，[6] の研究によりエラーが伝播する範囲は非常に小さいということがわかっている．エラー伝播が原因で生じたカーネルフェイラの約 90% は，エラーが発生したサブシステム内で完結をし，他のサブシステムにはエラーが伝播していない．Linux の kernel サブシステムでエラーが発生した場合，93.2% は 同じ kernel サブシステム内でのみエラー伝播起こしカーネルフェイラに至る．さらに，サブシステムのカーネルフェイラの約 60% が，フォールトの実行地点から 10 サイクル以内に発生している．

これらより，エラーが伝播する範囲はカーネルのごく一部であり，カーネル全体がエラー状態となってフェイラが

© 2018 Information Processing Society of Japan

発生するケースは少ない．

3. 提案

本論文ではプロセスローカル・エラーによるカーネルフェイラを検知しエラー状態を取り除くことで，カーネルの実行を継続する手法を提案する．プロセスローカル・エラーの場合，エラー状態がカーネル内のプロセスコンテキストに閉じているため，フェイラを起こしたプロセスのみを強制終了させることで，エラー状態のプロセスコンテキストをカーネルから取り除くことができる．

3.1 カーネル内部のデータ領域の再構成

プロセスローカル・エラーを検知するために，カーネル内部のデータ領域を再構成する．本論文では，次のように，カーネル内部のデータをプロセスローカル・データとカーネルグローバル・データの二つに分離する．

プロセスローカル・データ: 単一のプロセスコンテキストによってのみ使用されるデータを指す．データの代表的な例としては，ローカル変数やプロセス毎に使用されるレジスタなどを管理しているカーネルスタックがある．カーネルスタックはそれぞれのプロセスコンテキスト毎に確保され使用されるため，あるプロセスコンテキストが他のプロセスコンテキストのカーネルスタックを参照することはない．

カーネルグローバル・データ: 全てのプロセスコンテキストから参照され使用されているデータを指す．データの代表的な例としては，バッファ・キャッシュがある．このバッファ・キャッシュは全てのプロセスコンテキストが共有して使用しているデータである．そのため，ロックを使用してプロセスコンテキスト間で同期をとる必要がある．

プロセスローカル・エラーを検知するためには，カーネルグローバル・データに対して書き込みがあったかどうかを判断する必要がある．そのため，プロセスコンテキストが実行中には，カーネルグローバル・データを読み出し専用にしておく．これにより，プロセスコンテキストがカーネルグローバル・データに対して書き込みを行なったとき保護違反が発生するため，この違反を補足することで書き込みを検知し，書き込みの有無を知ることができる．

上記で述べたように，プロセスローカル・データは自身のプロセスコンテキストによってのみ使用されるデータである．そのため，自身のプロセスローカル・データが他のプロセスコンテキストから書き込まれる操作は望ましくない．あるプロセスコンテキストがカーネル内に潜んでいるフォールトを実行し，他のプロセスコンテキストのプロセスローカル・データに対し書き込みを行なったとする．その場合，そのプロセスローカル・データは期待とは異なった状態となり，他のプロセスコンテキストからエラーが伝播したことになる．このエラー伝播を防ぐため，プロセスローカル・データは自身のプロセスコンテキストからのみ書き込み可能とし，他のプロセスコンテキストからは書き込み禁止とする．プロセスローカル・データに対して隔離された環境を与えることで，エラー伝播の拡大を防ぐことができる．

3.2 対象とするフェイラ

一般のフェイラと同様にカーネルのフェイラも，エラー状態の伝播によってカーネルが停止するフェイルストップと，カーネルが停止せずに誤った結果を返すビザンチン・フェイラがある．本論文が対象とするフェイラは，カーネルが停止するフェイルストップを対象とする．本論文で示す手法は，従来のカーネルでは停止していたような場合であっても，安全にエラーが回復できる場合に限り，エラー回復を行うことを目的とし，ビザンチン・フェイラは対象としない．従来のカーネルでビザンチン・フェイラとなるようなエラー伝播に対しては，同じようにビザンチン・フェイラとなる．

あるフォールトからエラー伝播が発生し，カーネルが停止したとしよう．このとき，プロセスローカル・エラーによるフェイラであると保証できれば，フェイラを引き起こしたプロセスを強制終了することで，カーネル内のプロセスコンテキストも廃棄し，エラー状態から回復することができる．プロセスローカル・エラーであると保証できない場合には，保守的にカーネルグローバル・エラーとみなし，これまでと同じようにカーネルを停止させる．

フェイラによってカーネルが停止した際，それがプロセスローカル・エラーによるものなのか，カーネルグローバル・エラーによるものなのかを判別することは容易ではない．3.3 節で述べる手法では，1) プロセスローカル・エラーであると保証できるケース，2) プロセスローカル・エラーであるかカーネルグローバル・エラーであるか判定できないケース，3) カーネルグローバル・エラーであることの検出が遅れるケースの 3 つが存在する．

図 3 a) にプロセスローカル・エラーと判断できるケースを示す．あるプロセスコンテキストでカーネルを実行中にカーネルが停止したものの，そのプロセスコンテキストではカーネルグローバル・データへの書き込みを行なっていない場合である．この場合，そのプロセスコンテキスト内で発生したエラーが，カーネルグローバル・データに伝播していないことが保証できるため，フェイラを起こしたプロセスを強制終了すれば，カーネル内のデータの一貫性は保たれている．なお，このような場合であっても，カーネルグローバル・データにエラーがあり，それを読み込んだためにカーネルが停止したケースが考えられる．このケースについては，図 3 c) で議論する．

図 3 b) にプロセスローカル・エラーかカーネルグローバル・エラーか判別が難しいケースを示す．あるプロセス

© 2018 Information Processing Society of Japan

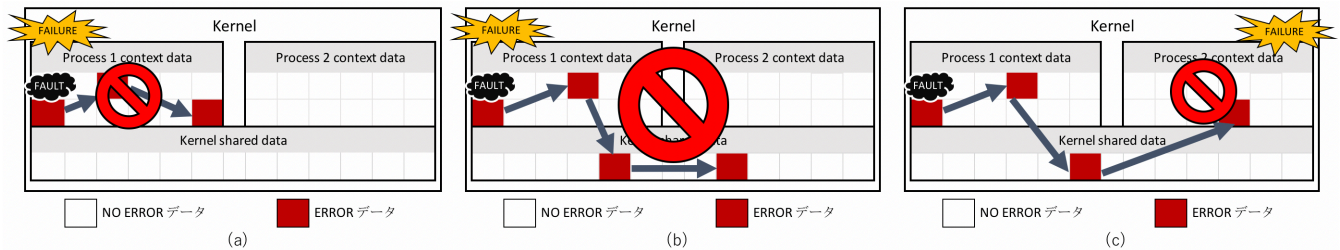

図 3　対象とするフェイラ

コンテキストでカーネルが停止したとしよう．そのプロセスコンテキストがすでにカーネルグローバル・データに書き込みを行なっていた場合，プロセスコンテキスト内のエラーがカーネルグローバル・データに伝播している可能性ある．このような場合，カーネルグローバル・データにエラーが伝播しているかどうか判別するのは容易ではないため，保守的にカーネルグローバル・エラーであるとみなし，カーネル全体を停止させる．

図 3 c) にカーネルグローバル・エラーの検出が遅れる例を示す．あるプロセスコンテキストでカーネルグローバル・エラーが発生し，カーネルグローバル・データにエラーがあるとしよう．そして，このエラーがすぐにフェイラとはならず，カーネルグローバル・データに残った状態になっていたとする．このような場合，あるプロセスコンテキストがそのカーネルグローバル・データに残ったエラーを読み込み，カーネルの実行が停止する場合がある．このような場合，そのプロセスコンテキストがカーネルグローバル・データへの書き込みを行なっていなければ，図 3 a) と同じ状態となり，一見したところ，プロセスローカル・エラーであると誤認識され，より深刻なエラー状態を引き起こすように見える．

しかし，図 3 c) のような場合，フェイラを起こしたプロセスコンテキストはカーネルグローバル・データへの書き込みを行なっていないため，すでにカーネルグローバル・データに存在するエラーがさらに伝播しているわけではない．すなわち，図 3 c) のようなフェイラが発生してもカーネルグローバル・エラーがさらに伝播することはなく，エラー状態のデータが増えるわけではない．したがって，このようなエラー伝播をプロセスローカル・エラーであると意図的に誤認識させても，エラーがさらに伝播することはない．図 3 c) のようなケースを図 3 a) のケースと厳密に区別しようとすると，プロセス・コンテキストがどのようなデータを読み込んだかを追跡する必要があり，その区別は容易ではない．したがって，本論文で提案するエラーの回復手法では，図 3 c) のようなケースを意図的にプロセスローカル・エラーであると誤認識させている．このようにしてもエラー伝播が広がる訳ではなく，従来のカーネルよりも深刻なエラーが生じることはない．

なお，この方式をとった場合，エラー状態のカーネルグローバル・データを読み込んだプロセスが繰り返しフェイラを起こす可能性がある．そのため，本論文で提案するエラーの回復手法ではプロセスローカル・エラーが頻繁に起きる場合には図 3 c) のケースが起きていると判断するようにしている．その結果，カーネルグローバル・エラーの検知が遅れることとなる．この点については 3.4 節で議論する．

3.3　エラーの検知手法

プロセスローカルエラーとカーネルグローバルエラーを区別して検知するために，カーネルグローバルデータに対して書き込みがあったかどうかを確認する必要がある．3.1 節で述べた，カーネルグローバル・データへの書き込みの検知を使用して，カーネルグローバル・データに書き込みがあったことを記録する．カーネルフェイラ時にこの記録を確認することでカーネルグローバル・データの書き込みの有無を判断し，エラーがプロセスローカル・エラーであるのか，カーネルグローバル・エラーであるのかの検知をすることが可能となる．この場合，一度でもカーネルグローバル・データに対して書き込みを行なった場合，正常にデータを更新できていたとしても，それ以降のフェイラは全てカーネルグローバル・エラーとして検知されてしまう．しかし，カーネルグローバル・データに書き込んだプロセスコンテキストが正常終了した場合，そのプロセスコンテキストが新たにカーネグローバル・データをエラー状態にしていることは考えにくい．理由としては，もし，そのプロセスコンテキスがカーネルグローバル・データにエラー状態を引き起こしているのであれば，2.1 節で述べたように，フェイラの多くはエラー状態から 10 サイクル以内に発生するため，正常終了する前にフェイラが発生すると考えられるからである．

そのため，カーネルグローバル・データへの書き込みフラグをそれぞれのプロセスコンテキストに毎に所持させ，プロセスコンテキスト単位でカーネルグローバル・データへの書き込みを記録するようにする．これにより，一度カーネルグローバル・データに対して書き込みを行なっていたとしても，そのプロセスコンテキストが正常終了した場合はフラグを廃棄し，再度プロセスローカル・エラーを検知することが可能となる．しかし，正常終了したプロセ

スコンテキストがカーネグローバル・データに対してサイレントエラーを生じさせている場合．正常終了した段階でフラグを廃棄するのは危険であるかのように見える．ここでいう．サイレントエラー状態とは，プロセスコンテキストが正常終了するまでフェイラに至らないエラー状態のことでを指す．これは 3.2 節で述べた図 3 c) のケースに対応する．このようなケースは擬似的にプロセスローカル・エラーとみなすことで．フラグを廃棄することができる．

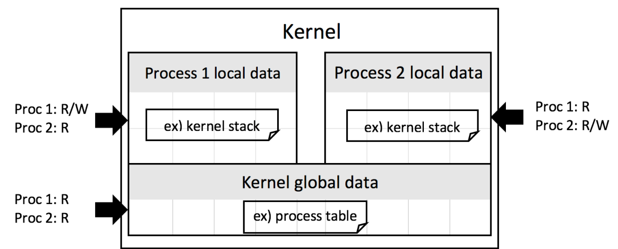

図 4　カーネル内部のデータ

3.4　カーネルグローバル・エラーの検出遅延

図 3 c) のケースが発生した場合，従来のカーネルと比べるとカーネル内にエラーが残る期間は大きくなる．しかし，2.1 節でも述べたように，カーネルフェイラの約 90% はエラーが発生したサブシステム内で完結し，その 60% はフォールトの実行地点から 10 サイクル以内に発生している．そのため，カーネルグローバル・データがエラー状態となった場合は，すぐにフェイラに至ることが考えられる．よって，図 3 c) のケースようにエラー状態がカーネルに残り続け，そのエラー状態を読み出すことでプロセスローカル・エラーが何度も発生することは考えにくい．もし，プロセスローカル・エラーが頻発して発生する場合は，カーネルの状態が図 3 c) のケースであると判断し，意図的にカーネルグローバル・エラーするなどの対応を取ることで，実用上困ることはないと考えられる．

4.　実装

提案した機構をマサチューセッツ工科大学が開発しているリサーチカーネルである xv6 [7] に実装した．本論文では提案手法の機構は shell プロセス以降に生成されたプロセスを対象とする．また，プロセスローカル・データとして確保するデータはカーネルスタック，ファイル構造体，パイプ構造体とする．

4.1　カーネル内部のデータ分離

カーネル内部のデータをプロセスローカル・データとカーネルグローバル・データの二つに分離するため，カーネルの仮想空間のヒープ領域をプロセスローカル・エリアとカーネルグローバル・エリアの二つに分ける．この分離はカーネルのブート時に行う．カーネルグローバル・データへの書き込みの検知と，プロセスローカル・データの隔離のために，図 4 のように，それぞれのデータに保護機構をつける．プロセスローカル・データは自身のプロセスのみ書き込みが可能であり，他のプロセスからは書き込み不可である．カーネルグローバル・データはプロセスの実行前に読み出し専用のデータとして書き込み不可にする．そのため，カーネルグローバル・データに書き込みがあった場合，ページフォールトが発生する．このページフォールトを捕捉し，書き込み処理を行なったプロセスが所持しているカーネルグローバル・データの書き込みフラグをセットする．その後，そのデータを書き込み可能に変更し動作を継続させる．

自身のプロセスローカル・データに対して他のプロセスから書き込み処理があった場合，ページフォールトが発生する．フォールトを起こしたデータのアドレスがプロセスローカル・エリアであった場合，不正な書き込みが行われたとしてフォールトを起こしたプロセスをキルする．

4.2　フェイラ時の処理

カーネルフェイラが発生した場合，パニック関数内でフェイラを起こしたプロセスの書き込みフラグを確認する．書き込みフラグがセットされていない場合は，プロセスローカル・エラーによるフェイラと判断できるため，しプロセス強制的に exit する．書き込みフラグがセットされている場合は，カーネルグローバル・エラーによるフェイラとし，通常のカーネルパニックと同様にカーネルの動作を停止する．

4.3　書き込み可能にするカーネルグローバルデータ

カーネルスレッドのカーネルスタックなどの，カーネルの実行に必要不可欠なカーネルグローバル・データを書き込み禁止にすると，プロセスの開始直後にカーネルはクラッシュしてしまう．そのため，このようなカーネルグローバルデータはあらかじめ書き込み可能にしておく必要がある．xv6 ではカーネルスレッドのカーネルスタックの他に 2 つ書き込み可能にしておく必要のあるカーネルグローバル・データが存在する．これらのデータをあらかじめ書き込み可能にしておくことで，カーネルはクラッシュせずに動作を継続させることが可能である．しかし，本提案のプロセスローカルエラーの正確性を保証することができなくなる．これらのデータに対して書き込みがあった場合，書き込みフラグはセットされない．そのため，プロセスローカルエラーと検知した場合でも，実際にはカーネルグローバル・エラーである可能性が考えられる．この問題を解決するため，プロセスローカルエラー時にこれらのデータがエラー状態でないか Verification する必要がある．Verification を行いデータがエラー状態であった場合は，カーネルグローバル・エラーとしてカーネルの動作を

表 1 実験環境

項目	詳細
ホスト OS	Ubuntu 16.04.03 LTS
ホスト CPU	Intel Core i7-4702HQ CPU @ 2.20GH
ホスト Memory	8GB
ホスト QEMU	v2.50
ゲスト OS	xv6
ゲスト CPU	1 VCPU
ゲスト Memory	512MB

停止させる．これにより，プロセスローカル・エラーの正確性を保証するとが可能となる．

本実装では 書き込み可能にしたカーネルグローバル・データに対しては，可能な限り Verification を行なっているが，完璧に Verification を行えていないデータも存在する．

4.4 割り込みハンドラによるカーネルグローバルデータの書き込み

プロセスの開始直後に割り込み処理が実行されるため，割り込みハンドラ内でカーネルグローバル・データへの書き込みが発生する．そのためプロセスの処理の開始と同時に書き込みフラグがセットされる．この書き込みフラグのセットによりそれ以降のプロセスローカル・エラーは全てカーネルグローバル・エラーとなってしまう．この書き込みフラグのセットは全てのプロセスの開始直後に共通して発生するため，プロセスローカル・エラーを検知することができなくなる．そのため，割り込みハンドラの処理は専用のページテーブルを割り当て，カーネルグローバル・データへの書き込みを全て可能にする．そして，割り込みハンドラでの処理中にフェイラが発生した場合は，カーネルグローバル・エラーとする．

5. 実験

提案したカーネルがプロセスローカル・エラーを検知し動作を継続できることを確認するため，ソフトウェアフォールトインジェクション (SFI) による実験を行なった．実験環境を表 1 に示す．

実験は文献 [8] で提案された SFI 技術である G-SWIFT が挿入するフォールトの種類に基づき手動で行なった．G-SWIFT は，実際のソフトウェアのバグの調査において高い出現率を占めたフォールトのみを挿入するように実装されている．そのため G-SWIFT が挿入しているフォールトの種類を今回の実験では利用した．通常の xv6 と本提案の機構を導入した xv6 に それぞれ同じ 20 件のフォールトを手動で挿入した．その実験結果を表 2 に示す．

本提案を実装した xv6 では，挿入した 20 件のフォールトの内，7 件のフォールトによって発生したエラーをプロセスローカル・エラーとして検知をし，プロセスのキルを行うことでカーネルの実行継続を可能にした．通常の xv6 では，この 7 件のフォールトによるエラーは全てカーネルフェイラを引き起こした．この結果から，プロセスローカル・エラーはフェイラを起こしたプロセスを強制終了させることで回復可能であることがわかった．

しかし，本提案を実装した xv6 でクラッシュした 13 件のフォールトの内 7 件は，プロセスローカル・エラーとして検知をしてプロセスのキルを行なったが最終的にカーネルフェイラとなったものであった．このカーネルフェイラに至った原因は，プロセスの強制終了によるデットロックの発生であった．例えば，パイプ構造体に対してロックの解放忘れによるフォールトを挿入したところ，パイプ通信でデットロックが発生しカーネルが停止した．この結果から，パイプ構造体を参照してるプロセスグループ全体をキルする機構が必要であることがわかった．また，プロセスグループ全体をキルした上で，そのプロセスが獲得していたロックを解放する機構も必要であることがわかった．しかし，xv6 ではプロセスグループの概念がないためこの機構を導入することができなかった．

6. 関連研究

カーネルの障害から効率的にリカバリーすることで，システムの信頼性を向上させる様々な手法が研究されている．Otherworld [9] ではカーネルフェイラが発生した場合アプリケーションの実行状態を失うことなく OS を再起動する手法を提案している．カーネルフェイラが発生しカーネルを再起動した後，フェイラが発生した時点でのアプリケーションのメモリ空間やオープンしているファイルなどの状態を復元する．しかし，Otherworld は再起動によってアプリケーションのリカバリーを行うため，システムのダウンタイムは避けられない．本論文で提案したカーネルでは再起動をすることなくアプリケーションの実行を継続することができるため，システムのダウンタイムの発生を回避することができる．

カーネルのコンポーネントに焦点を当てたカーネルフェイラからのリカバリー手法には Nooks [10] がある．Nooks ではデバイスドライバのフェイラを OS から隔離し，OS の信頼性を向上させる手法を提案している．Nooks はメモリ保護を使用してデバイスドライバからカーネルを Isolation し，ドライバのフェイラがカーネルに影響を及ぼす前に，ドライバのフェイラを検知しリカバリーを行う．また，Swift らは Nooks のアプローチを使用してデバイスドライバに対する耐障害性を向上するメカニズムである Shadow Driver [11] を提案している．Shadow Driver はデバイスドライバを監視し，ドライバの障害から透過的にカーネルをリカバリーすることが可能である．しかし，これらの手法は拡張ソフトウェアによるフェイラからのリカバリー手法であり，カーネルそのものがフェイラに陥った場合，リカバリーすることはできない．本論文で提案した手法では，カーネル

表 2 実験結果

カーネル	挿入したフォールトの数	リカバリーした件数	クラッシュした件数
提案を実装した xv6	20	7	13
通常の xv6		0	20

自身のフェイラからエラー状態を取り除くことで，カーネルの動作を継続させることが可能である．

仮想化を有効に活用しシステムの信頼性を向上させる手法には CuriOS [12] がある．CuriOS はマイクロカーネルベースの OS で，クラッシュしたサービスを透過的にクライアントにリカバリする．CuriOS はクライアントのコンテキストをクライアントからアクセスできないメモリに保存する．そして，システムサーバの OS に障害が発生した場合，クライアントのコンテキストを失うことなくシステムの再起動を可能にする．また，VINO [13] は OS を再起動することなく拡張機能の障害からリカバリーするメカニズムを提供している．OSIRIS [14] では，システム状態の一貫性を保ったままリカバリーする手法を提供している．OSIRIS はシステムのフェイラ時にチェックポイントを使用して，システム状態の一貫性を失うことなくロールバックする．しかし，これらの手法はマイクロカーネルや特定のカーネルを対象としており，汎用 OS に適用することは難しい．本論文で提案したカーネルの設計は，Linux などの汎用 OS に適用することが可能である．

また，カーネルの設計および実装の正確性を形式的検証に基づき OS の堅牢生を向上させる手法が研究されている．seL4 [15] はモデル検査を用いて，カーネルの機能の正確さを保証したマイクロカーネルである．これにより，seL4 はカーネルが安全な操作のみを実行し，フェイラを引き起こさないことを保証している．Hyperkernel [16] では，SMT ソルバを使用して実装および検証を行なったカーネルである．xv6 をベースに検証および実装を行い，xv6 よりもバグが少ない Hyperkernel の開発に成功している．しかし，形式的検証は設計と実装に非常にコストがかかってしまう．seL4 では 1 万行の C のソースコードを検証するのに，20 万行の検証コードが必要である．そのため，Linux などの汎用 OS にこの手法を適用することは困難である．

7. まとめ

本論文では，エラー状態がカーネル内のプロセスコンテキストに閉じるプロセスローカル・エラーを検知し，エラー状態を取り除くことでカーネルの実行の継続を可能にする手法を提案している．全てのプロセスコンテキストで共有して使用されているデータにエラーが伝播するカーネルグローバル・エラーと区別するために，カーネル内部のデータ領域をプロセスローカル・データとカーネルグローバル・データの二つに分離する．そして，プロセスローカル・エラーを検知する機構を導入し，フェイラを起こしたプロセスをキルすることでエラー状態を取り除く．これにより，従来のカーネルではフェイラとなっていた場合でも，カーネルを停止させずに他のプロセスの実行を継続することができる．

謝辞

本研究は，JST, CREST, JPMJCR1683 の支援を受けたものである．

参考文献

[1] Oppenheimer, D., Brown, A., Beck, J., Hettena, D., Kuroda, J., Treuhaft, N., Patterson, D. A. and Yelick, K.: ROC-1: hardware support for recovery-oriented computing, *IEEE Transactions on Computers*, Vol. 51, pp. 100–107 (online), DOI: 10.1109/12.980002 (2002).

[2] Patterson, D. A.: Recovery oriented computing: a new research agenda for a new century, *Proceedings Eighth International Symposium on High Performance Computer Architecture*, pp. 223–223 (online), DOI: 10.1109/HPCA.2002.995714 (2002).

[3] Palix, N., Thomas, G., Saha, S., Calvès, C., Lawall, J. and Muller, G.: Faults in Linux: Ten Years Later, *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, New York, NY, USA, ACM, pp. 305–318 (online), DOI: 10.1145/1950365.1950401 (2011).

[4] Guo, L., Senna Tschudin, P., Kono, K., Muller, G. and Lawall, J.: Oops! What about a Million Kernel Oopses?, Technical Report RT-0436, INRIA (2013).

[5] Yoshimura, T., Yamada, H. and Kono, K.: Is Linux Kernel Oops Useful or Not?, *Presented as part of the Eighth Workshop on Hot Topics in System Dependability*, Hollywood, CA, USENIX, (online), available from ⟨https://www.usenix.org/conference/hotdep12/workshop-program/presentation/Yoshimura⟩ (2012).

[6] Gu, W., Kalbarczyk, Z., Ravishankar, Iyer, K. and Yang, Z.: Characterization of linux kernel behavior under errors, *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.*, pp. 459–468 (online), DOI: 10.1109/DSN.2003.1209956 (2003).

[7] R. Cox, M. F. K. and Morris, R. T.: Xv6, a simple Unix-like teaching operating system, 2017, `http://pdos.csail.mit.edu/6.828/xv6`.

[8] Duraes, J. A. and Madeira, H. S.: Emulation of Software Faults: A Field Data Study and a Practical Approach, *IEEE Transactions on Software Engineering*, Vol. 32, No. 11, pp. 849–867 (online), DOI: 10.1109/TSE.2006.113 (2006).

[9] Depoutovitch, A. and Stumm, M.: Otherworld: Giving Applications a Chance to Survive OS Kernel Crashes, *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, New York, NY, USA, ACM, pp. 181–194 (online), DOI: 10.1145/1755913.1755933 (2010).

© 2018 Information Processing Society of Japan

[10] Swift, M. M., Bershad, B. N. and Levy, H. M.: Improving the Reliability of Commodity Operating Systems, *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, New York, NY, USA, ACM, pp. 207–222 (online), DOI: 10.1145/945445.945466 (2003).

[11] Swift, M. M., Annamalai, M., Bershad, B. N. and Levy, H. M.: Recovering Device Drivers, *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, Berkeley, CA, USA, USENIX Association, pp. 1–1 (online), available from ⟨http://dl.acm.org/citation.cfm?id=1251254.1251255⟩ (2004).

[12] David, F. M., Chan, E. M., Carlyle, J. C. and Campbell, R. H.: CuriOS: Improving Reliability Through Operating System Structure, *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, Berkeley, CA, USA, USENIX Association, pp. 59–72 (online), available from ⟨http://dl.acm.org/citation.cfm?id=1855741.1855746⟩ (2008).

[13] Seltzer, M. I., Endo, Y., Small, C. and Smith, K. A.: Dealing with Disaster: Surviving Misbehaved Kernel Extensions, *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, New York, NY, USA, ACM, pp. 213–227 (online), DOI: 10.1145/238721.238779 (1996).

[14] Bhat, K., Vogt, D., v. d. Kouwe, E., Gras, B., Sambuc, L., Tanenbaum, A. S., Bos, H. and Giuffrida, C.: OSIRIS: Efficient and Consistent Recovery of Compartmentalized Operating Systems, *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 25–36 (online), DOI: 10.1109/DSN.2016.12 (2016).

[15] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H. and Winwood, S.: seL4: Formal Verification of an OS Kernel, *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, New York, NY, USA, ACM, pp. 207–220 (online), DOI: 10.1145/1629575.1629596 (2009).

[16] Nelson, L., Sigurbjarnarson, H., Zhang, K., Johnson, D., Bornholt, J., Torlak, E. and Wang, X.: Hyperkernel: Push-Button Verification of an OS Kernel, *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, New York, NY, USA, ACM, pp. 252–269 (online), DOI: 10.1145/3132747.3132748 (2017).

© 2018 Information Processing Society of Japan