情報処理学会研究報告
IPSJ SIG Technical Report

Vol.2011-ARC-195 No.6
Vol.2011-OS-117 No.6
2011/4/13

オペレーティングシステムカーネルにおけるエラー伝播の調査

吉村 剛[†1] 山田浩史[†1,†2]
吉田哲也[†1] 河野健二[†1,†2]

コンピュータには高い信頼性が求められている．信頼性を脅かす要因としてオペレーティングシステムのカーネルフェイラがある．カーネルフェイラが発生すると，カーネル上で動作している全てのアプリケーションの動作異常につながり，サービス提供者やユーザに大きな損害を与えてしまう．カーネルフェイラの原因はソフトウェアやハードウェアにおけるバグであり，こうしたバグを開発段階で除去することは難しい．そのため，カーネルフェイラが運用時に発生したらできるだけ早くフェイラから復旧する方法が求められている．カーネルフェイラから素早く復旧する手法を検討するために，本研究はバグの発生からフェイラに至る過程で発生するエラー伝播を調査する．実用 OS のバグの調査結果に基づいて作成されたフォールトインジェクタを利用して，Linux カーネルに対してフォールトインジェクションを行い，その結果発生するエラー伝播を KDB によって追跡した．調査した結果，フォールトインジェクション 500 回に対してエラーを 150 回，エラー伝播を 18 回確認した．エラー伝播はプロセス間では発生しにくく，バグの挿入された関数内で収束する確率が非常に高いことを確認した．

Error Propagation on a Commodity Operating System Kernel

Takeshi Yoshimura,[†1] Hiroshi Yamada,[†1,†2]
Tetsuya Yoshida[†1] and Kenji Kono [†1,†2]

Kernel failures have a considerable impact on the overall availability of software systems. Even if the applications running on the operating system are highly available, a bug inside the kernel may result in a failure of the entire software stack. However, modern operating systems are far from bug-free. Guided by this fact, we need to recover from kernel failures as quickly as possible when they occur in a service operation. To explore the quick recovery mechanism, in this work, we investigate error propagation of kernel bugs in a commodity operating system kernel. In our investigation, we use a fault injector that is implemented based on a survey of commercial operating system bugs. We inject faults into Linux 2.6.18.8 with the injector, and trace error propagation with KDB. Our experimental result shows that 150 of 500 faults make the kernel error, which means that almost all the error is not propagated. Our result also shows that 18 of 150 errors are propagated, and the errors are not propagated from the faulted process to the other.

1. はじめに

コンピュータには高い信頼性が求められている．サービスが停止すると，その経済的損失は大きいためである．ハイエンドサーバの停止は 1 時間ごとに 10 万ドルの損失につながると算出されている[1)]．また，サーバの運用コストのうち，障害からの復旧やその対策のための費用が 30%から 50%という大きな割合を占めている[2)]．

コンピュータシステムの信頼性を脅かす要因の 1 つとして，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．カーネルフェイラとは，カーネルにおいて致命的なエラーが発生したために，本来の動作が不可能になった状態を指す．カーネルのクラッシュやハングはフェイラのひとつである．カーネルフェイラが起きると，その上で稼働している全てのアプリケーションの動作異常につながり，サービス提供者やユーザにとって大きな損害となる恐れも大きい．

カーネルフェイラが発生する原因としてソフトウェアやハードウェアにおけるバグが挙げられる．こうしたバグを開発段階で除去することは難しく，カーネルフェイラがサービス運用時に発生することは避けられない状況にある．近年 OS カーネルは複雑になる傾向にあり，カーネルコード内でのバグを開発段階で完全に取り除くことを難しい[3),4)]．また，例えばメモリ内の特定ビットが固定されたりするなど，ハードウェアにおいて発生するバグも存在する[5)]．他にも外的な要因で一時的にメモリがビット反転することもあることが知られている[6)]．

本研究では，バグの発生からフェイラに至る過程で発生するエラー伝播を調査する．エラー伝播とは，あるエラー状態にあるデータを用いて計算した結果，別のデータも異常値になり，新たにエラーが発生する一連の流れを指す．エラー伝播の結果，システムがフェイラ

†1 慶應大学
Keio University
†2 JST CREST

Vol.2011-ARC-195 No.6
Vol.2011-OS-117 No.6
2011/4/13

に至る．カーネル内のバグによるエラー伝播を調査することで，フェイラ時のメモリの状態からエラーの傾向をつかみ，カーネルフェイラからの高速リカバリ手法の足掛かりを探す．

カーネルバグのエラー伝播を調べるために，本研究では，実用 OS のバグの調査結果に基づいて作成されたフォールトインジェクタを Linux 2.6.18.8 に用いた．使用したフォールトインジェクタによって提供される 10 種類すべてのバグを Linux 2.6.18.8 に挿入し，バグ挿入によって発生するエラー伝播を KDB を用いて追跡した．

実験結果により次のことがわかった．エラー伝播はプロセス間では発生しにくく，バグの挿入された関数内で収束する確率が非常に高い．頻繁なエラーチェックやそれに伴うフェイルストップによって，エラーが関数やプロセス内でアイソレーションされる傾向にあったためである．一方で，OS カーネルは本調査で用いるフォールトのような，メモリを直接変化させるフォールトに対して耐性がないこともいえる．そのため，メモリハードウェアエラーに対して脆弱である．しかし，メモリハードウェアの異常によって発生するエラー伝播でもプロセスや関数内でアイソレーションされる傾向にあることや，カーネルがエラー検知が可能であること利用して，今後高速リカバリ手法を考えていくことになる．

本論文の構成を以下に示す．2 章でカーネルにおけるフォールトとフェイラの一種であるバグとクラッシュを対象とした関連研究をそれぞれ紹介する．3 章ではエラー伝播の調査方法として用いるフォールトインジェクションについて説明する．4 章で実際の調査の方法について説明し，その結果を 5 章で述べる．6 章で結論を述べ，最後に今後の課題を述べる．

2. 関 連 研 究

ソフトウェアやハードウェアにおけるバグの調査は多くなされてきた．2001 年に Linux の 21 種類の異なるバージョン (1.0 から 2.4.1) においてソースコードを解析することで，バグの発生する傾向や，バグが修正される傾向を調査したところ，デバイスドライバにおけるバグが大きな割合となることがわかった[3]．しかし 2011 年にはその傾向が変化しつつあり，アーキテクチャ依存コードの部分にバグが含まれる割合が大きくなっていることがわかっている[4]．また，メモリハードウェアにおけるバグとそのバグがソフトウェアに及ぼす影響についての調査では，800GB のメモリを 9 か月間観察したところ，トランジエントなエラーよりもそうでない永続的なエラーが多く見られることがわかった[5]．他にも宇宙線などの外的な要因により，メモリハードウェアにバグがなくとも偶発的にビット反転がおきることがあることも知られている[6]．

古くからシステムのクラッシュについての分析も多くなされてきている．1990 年のタンデムコンピュータにおける調査で，発生したフェイラの 55%がソフトウェアが原因であることが調査結果として得られている[7]．同様に 1993 年にもタンデムコンピュータのフェイラ時に出る実行ログの解析をすることで，エラー伝播のモデル化もなされている[8]．1991 年にも同様に， MVS オペレーティングシステムのログを利用してエラーのモデル化がなされている[9]．最近の研究では Windows XP のクラッシュ時のコアダンプを収集してクラッシュの原因の傾向を調査することで，多くのクラッシュはグラフィックデバイスドライバが原因となって起こることがわかっている[10]．

本研究では以上のようなフェイラの根本原因であるバグの分析や，システムフェイラ後のエラーやフェイラの分析を行うのではなく，エラー状況下のカーネルの挙動を分析する．それによってより本質的なカーネルエラーからのリカバリ手法を探ることが可能となる．

本研究と同様に，エラー状況下の Linux カーネルの挙動分析[11] や，エラー状況下の UNIX システムの挙動分析[12] が行われている．両者ともクラッシュに至るエラー伝播に注目している．前者では多くのエラーが短いサイクルでクラッシュに至っており，一部のエラーがサブシステム間で伝播していることがわかった．後者では統計を用いて，ハードウェアフォールトによるエラー伝播と比較して，ソフトウェアフォールトによるエラー伝播がクラッシュ発生までの時間が長いことを指摘している．

本研究ではシステム稼働中に発生するエラー伝播を最初のエラー発生から収束するまで追跡し，クラッシュやハングなどを含むフェイラに至るケースやエラーが訂正されるケースを含め全て追跡する．それぞれの結果に至った原因の分析をすることによって様々なエラー伝播の性質や，現在のカーネルのエラー耐性を知ることができる．

3. フォールトインジェクション

本研究では Linux カーネルにおけるフェイラの発生過程となるエラー伝播について調査する．そのために，カーネルに対してフォールトインジェクションをし，エラーの伝播を KDB を用いて観察する．ここでは本研究で用いるフォールトインジェクタである SWIFI(SoftWare Implemented Fault Injection) の詳細や，再現するフォールトについて述べる．

3.1 SWIFI

SWIFI はハードウェアバグや，プログラミングのミスによるソフトウェアバグ，合計 10 種類のバグを再現する．合計 10 種類のバグはシステムフェイラの原因を調査する既存研究において，フェイラの原因となったハードウェアやソフトウェアのフォールトを分類した際に得られたものである．具体的にはメインフレーム用 OS である MVS の調査[9] や，フォー

Vol.2011-ARC-195 No.6
Vol.2011-OS-117 No.6
2011/4/13

表 1 ソフトウェアバグを再現するフォールトによる開発コードの変更例

フォールト名	変更前	変更後
INIT	`int x=0;`	`int x;`
DST&SRC	`x += 1;`	`x += 2;`
BRANCH	`if (x != 0) x = 1 / x;`	`x = 1 / x;`
PTR	`ptr = list->prev;`	`ptr = list->next;`
LOOP	`if(x == 0)`	`if(x != 0)`
INTERFACE	`func(argv[0]);`	`func(argv[1]);`
IRQ	`local_irq_restore()`	削除

ルトトレラント OS であるタンデムコンピュータの調査[8] において，フェイラを引き起したフォールトをフェイラ時のメモリダンプや全実行ログから分類・整理したものを利用している．また，ソフトウェアベースのフォールトインジェクタの調査[12)–14)] で提案されている，ハードウェアフォールトをソフトウェアで再現する方法や，トランジエントなエラーを再現する方法を利用している．SWIFI は Rio File Cache[15] や Nooks[16] の他にも Shadow Driver[17]，Otherworld[18] など，様々なシステムの信頼性測定に利用されている．

SWIFI はカーネルのシステムコールルーチンとして実装されている．SWIFI はカーネルテキスト領域からバグを再現することの可能な命令を検索し，ターゲットとなった命令をバグのある命令と置換する方法によりフォールトインジェクションをする．カーネルテキスト領域にはカーネルソースコードをコンパイルして得られたバイナリがあるため，命令を適切に置換することによって，命令の位置と対応するカーネルソースコードに疑似的にバグを挿入することが可能になる．SWIFI は Nooks プロジェクトのサイト[19] で入手し，本研究で対象とする Linux カーネルのバージョン 2.6.18 でも利用できるように移植作業をした．移植作業ではバージョン間の差異による変数名や実装の違いを補完し，フォールトインジェクションのターゲットがカーネルモジュールだったものをカーネルコアに変更した．

3.2 再現するバグの詳細

SWIFI が再現するバグとその再現方法について述べる．SWIFI はハードウェアエラーによるメモリバグと，プログラミングミスによって発生するソフトウェアバグを再現する．ソフトウェアバグを再現するフォールトによる開発コードの変更例を表 1 に示す．

- TEXT FAULT
  ハードウェアのバグなどで起こるメモリのビット反転のうち，カーネルテキストに対するものを再現する．その方法として，カーネルテキストのランダムなアドレスの内容をビット反転する．
- STACK FAULT
  ハードウェアのバグなどで起こるメモリのビット反転のうち，カーネルスタックに対するものを再現する．その方法として，カーネルスタック領域のランダムなアドレスの内容をビット反転する．
- INIT FAULT
  初期化忘れを再現する．実際にはスタックベースポインタよりオフセットが負のアドレスへ即値を代入する命令を削除する．
- NOP FAULT
  分岐エラーやタイミング等の要因で実行すべきだった命令を実行できない状況を再現する．実際にはカーネルテキスト内の任意の命令を削除する．
- DST&SRC FAULT
  計算や条件文で用いる変数や定数の間違いを再現する．実際にはオペランドを表すビットをランダムに反転する．また，SWIFI は DST FAULT, SRC FAULT は別々のフォールトとしているが，実際には同じ処理をするので本研究では区別していない．
- BRANCH FAULT
  条件分岐エラーやエラー処理忘れを再現する．実際にはジャンプ命令を削除する．
- PTR FAULT
  ポインタのメンバ指定ミスによる代入エラーを再現する．実際にはソースやディスティネーションにアドレッシング部分があれば，そのオフセットをランダムにビット反転する．
- LOOP FAULT
  分岐の条件エラーを再現する．実際にはジャンプ命令を反転した命令と変更する．
- INTERFACE FAULT
  関数の引数エラーを再現する．実際にはスタックベースポインタよりオフセットが正のアドレスの内容を他のアドレスやレジスタにコピーする命令を削除する．
- IRQ FAULT
  割り込み禁止の解除忘れを再現する．その方法として，
  `local_irq_restore()` の呼び出しを削除する．本調査環境では，連続した 2 命令 `push reg, popf` を削除することで再現可能となる．フラグレジスタのリストアができず割り込み許可ビットが禁止状態のままとなる．

Vol.2011-ARC-195 No.6
Vol.2011-OS-117 No.6
2011/4/13

4. エラー伝播の調査方法

4.1 調 査 目 的

本調査では，Linux 2.6.18.8 に対するバグ挿入により発生するエラー伝播を追跡することで，エラー伝播の性質を明らかにすることを目的とする．そして，エラー伝播の波及範囲や因子となるデータの種類や Linux 2.6.18.8 のエラーに対する耐性を分析することで，エラー伝播対策への手掛かりを掴むことを目的とする．実際に広く利用されている Linux において調査することで，現実に発生するエラー伝播の性質が明らかになる．それによって現在一般的に普及しているオペレーティングシステムにおいても，カーネルフェイラからの効果的な高速リカバリ手法を考案することが可能となる．

4.2 調 査 方 法

SWIFI を用いて各フォールトそれぞれ 50 回ずつ合計 500 回フォールトインジェクションする．また，以下の手順で調査を行う．

(1) 命令の変化やメモリの変化をカーネルデバッガで観察する．
(2) フォールトとなった命令を実行させるために，全てのデーモンを再起動させる．タイマ割り込みで自動的に実行されるケースも同様に調査する．
(3) 実行してメモリの変化を観察する．
(4) 実行可能な場合は Linux のソースコードでのフォールトの位置を特定し，エラー伝播の因子や波及範囲を分析する．
(5) 1 回のフォールトインジェクションの調査が終了後，システムを再起動してメモリを健全な状態に戻す．SWIFI のフォールトはメモリ書き換えによって再現されるため，ハードディスクからカーネルイメージを再ロードすることでフォールトは消滅する．

以上が調査手順である．ただし，フォールトインジェクションが失敗した場合は調査の対象外とする．調査環境である i386 では命令が固定長ではないため，SWIFI が命令の起点ではないアドレスをターゲットにしてしまい，正しく変換できない場合もある．例えばイミディエイト命令を別の命令と誤認してしまった結果，本来のフォールトと異なる命令に変更してしまう場合がある．例外的に TEXT FAULT に関してはハードウェアバグを再現するものなので，置換したものは全て調査の対象としている．フォールトインジェクションの成功を確認するため，本調査では調査手順の最初にデバッガで命令が正しく置換されたか確認する．他にも調査で用いるカーネルデバッガ本体へフォールトインジェクションした場合も調査の対象外としている．

表 2 各フォールト 50 回の調査に対するエラー，エラー伝播，フェイラの総数

フォールト名	エラー数	エラー伝播数	フェイラ
TEXT	14	8	5
STACK	50	0	5
INIT	13	6	0
NOP	9	1	9
DST&SRC	5	0	4
BRANCH	13	0	1
PTR	10	1	4
LOOP	11	0	4
INTERFACE	8	2	5
IRQ	17	0	6
ALL	150	18	43

4.3 調 査 環 境

本調査で用いた環境は，Windows7 の VMWare Workstation 7.1.2 上でゲスト OS として稼働させている i386 上の Linux 2.6.18.8 である．メモリは 1GB ，CPU コア数は 1 個 (ホストの CPU は Intel(R) Core(TM)2 Extreme CPU Q9300 @ 2.53GHz 2.53GHz)，ハードディスク 20GB の仮想化環境で調査した．より現実的な環境を再現するため，カーネルコンパイル時のコンフィグはインストール時の構成を変更せずに利用した．

多くのシステムにおいてバグは存在している可能性があるが，調査対象とするデバイス環境及びカーネルにはバグが存在しないことを前提とする．カーネルが実行する式は全て正しい式であり，式による計算で得られるデータもエラーではないため，フォールトインジェクションの結果エラーになったデータがシステムで唯一のエラーとなる．

5. 調 査 結 果

5.1 各フォールトごとの結果

各フォールト 50 回の調査に対して得られたエラー，エラー伝播，フェイラの総数を表 2 に示す．フォールトインジェクション合計 500 回に対してエラーに至ったケースは 150 個，エラー伝播が確認できたケースが 18 個，エラーがフェイラに至ったケースが 43 個であった．

各フォールトごとの結果は次のようになった．STACK FAULT によるエラーは上書きやコンテキスト終了によって消滅する傾向にあり，フォールトが全てエラーになっているのはエラーの定義がデータが異常値であることとしているためである．ビット反転を行う TEXT，DST&SRC，PTR FAULT によって発生するエラーはメモリアクセス違反を引き

Vol.2011-ARC-195 No.6
Vol.2011-OS-117 No.6
2011/4/13

起してクラッシュに至るケースが多く見られた．TEXT FAULT では，ビット反転によって命令の起点が破壊された結果数個に渡る命令の置換が発生したため，致命的なエラーを引き起しやすい傾向にあった．BRANCH FAULT ではエラー処理を削除することが多かったため調査時の環境ではフェイラは発生しにくく，LOOP FAULT では異常のない変数をエラーと扱ってしまい，不要なコードを実行するケースが多く見られた．NOP FAULT では多くのフォールトがスケジュール関数に挿入されたため，全てのエラーがアプリケーションの動作に大きく影響し，フェイラを引き起していた．IRQ FAULT はエラー伝播は伴わないが，多くのフォールトはメモリ管理やスケジュール関数に挿入されており，全てのフェイラはフェイルストップによって発生していた．

エラー伝播は全般的に，ローカル変数や関数引数・返値のような関数内変数を介して発生していた．グローバル変数やヒープに動的確保された変数のような，プロセス間で共有するようなデータはあまりエラー伝播の因子とならなかった．また，多くのエラーは原因となったフォールトの挿入された関数内で収束する傾向にあり，プロセス間でエラー伝播をすることはみられず，関数間で関数引数や返値を介して発生する傾向がみられた．

5.2 エラー伝播の性質

本調査において，多くのエラー伝播は同一プロセスに呼び出された関数間で発生し，プロセス間でエラーが伝播することは確認されなかった．プロセス間のエラー伝播を引き起す可能性があるのはプロセスで共有するデータをエラーにするフォールトが挿入される場合である．予備調査でネットワークデバイスドライバへのフォールトインジェクションをテストしている中で，ヒープへのコピーがオーバーランしてネットワーク用のデータ構造とは無関係なデータ構造を破壊し，その後暫くしてフェイラに至るケースがあった．しかし，カーネルコアを対象とする本調査ではこのようなケースは得られなかった．

エラー伝播がプロセス内のデータで収束する傾向にあるのは，多くのエラーはローカル変数や関数引数，返値などの関数内スコープにある変数により伝播していたためである．関数内の変数において発生したエラーは関数内でアイソレーションされるため，発生したプロセス内で収束する結果になったと考えられる．関数内スコープ以外の変数として，ヒープに動的確保された変数を介してエラー伝播するものがあげられる．本調査であった典型的な例をあげると， INTERFACE FAULT を挿入したケースでメモリディスクリプタを管理している木構造を崩してしまうものであった．しかし，最後まで 1 つのプロセスがエラーとなったデータを操作するだけで，他のプロセスに影響を及ぼさなかった．その原因はそのデータ構造に対してロックがかけられていたため，他のプロセスがそのデータにアクセスできない状態にあったためである．このように，グローバル変数やヒープ上の変数など，全てのプロセスがアクセス可能なデータに関しても，排他制御によりそのデータが他のプロセスからアイソレーションされていることが多いと考えられる．以上より，エラー伝播は因子となった変数に関らず，他のプロセスへ伝播する前に最初にエラーとなったプロセス内で収束しやすい傾向にあるといえる．

エラーがその原因となったフォールトのある関数やその近くで呼び出された関数で収束する傾向にあったのは，エラーチェックが頻繁に行われていたためである．エラー伝播の因子となるのはほとんどは関数呼び出しの際の引数や返値であったが，Linux は関数の呼び出し前後に引数や返値のチェックが多くなされており，そこで適正な値に戻すことでクラッシュを防いだり，フェイルストップすることでエラーのこれ以上の拡散を防いでいた．エラーチェックは本来タイミング等の問題で発生するエラーや，ハードウェアの故障で発生するクラッシュを防ぐ目的で行われることが多いが，プログラミングミスによるエラーを防ぐことにもつながっていた．また，多くの引数や返値は関数内で頻繁に使用されるため，致命的なエラーを引き起しやすい傾向にあり，クラッシュによってエラー伝播が収束しているケースも多く見られた．同様に，フォールトによって発生する最初のエラーの多くは直後で頻繁に使用される傾向にあり，クラッシュを引き起してエラー伝播が収束するケースが多かった．このように，多くのエラーは関数呼び出しの前後，もしくはフォールトの地点の直後に収束する確率が高いといえる．以上より，エラーはフォールトの挿入された関数で収束する確率が一番高く，その後は呼び出される関数の順に収束する確率が高くなっていく傾向にあるといえる．

Linux カーネルはクラッシュが起きるとカレントプロセスを消滅させる．その結果，カーネルスタックも消滅するので，ほとんどのエラー伝播は消滅すると考えられる．それにはひとつのサービスが犠牲となる問題点と，それ以外のデータのエラーは訂正しないため根本的なフェイラの原因解決には至っていないという問題点がある．しかし，排他制御や引数・返値のエラーチェックなどの本来バグやエラー伝播の対策手法でないものが，エラー伝播の広範囲に及ぶ拡散やそれに伴うカーネルクラッシュを予防するために働いていた．そのため，Linux カーネルにおけるエラー伝播は 1 つのプロセスや関数の中で限定的に発生しており，エラーは適切にアイソレーションされる傾向にあると考えられる．

5.3 エラー伝播の訂正・検知

本調査では，150 件あるエラーのうち 43 件がフェイラに至っており，Linux はエラーを訂正できていないことがわかる．特にハングやカーネルサブシステムの動作異常を防げてい

Vol.2011-ARC-195 No.6
Vol.2011-OS-117 No.6
2011/4/13

表 3 BRANCH FAULT の具体例

ターゲット	_page_cache_release+0x28
コードの場所	include/asm-generic/bug.h の 15 行目
元の命令	jne 0xc104ecb9 __page_cache_release+0x32
変化後の命令	nop nop
元のコード	#define BUG_ON(condition) do { if (unlikely((condition)!=0)) BUG(); } while(0)
変化後のコード	#define BUG_ON(condition) do { BUG(); } while(0)
エラー伝播	なし

ないことが多い．開発段階のバグがないという前提においても，メモリハードウェアの故障や外的要因で発生するようなメモリのビット反転によって引き起されるフェイラは防げないことがわかる．そしてそれは，ソフトウェアに原因があるように見えてしまうのも問題である．メモリハードウェアのエラーやそれに伴うエラー伝播に対して，Linux カーネルは訂正することは難しいのが現状となっており，メモリハードウェアエラーに対して脆弱であるといえる．

一方で，カーネルはエラーチェックを頻繁に行うことで，カーネルクラッシュやカーネルパニックによるシステム全体の動作異常を出来る限り回避していた．そのため，フェイラと判定したケースでも，デバイスが使用不可能になるケースや，デッドロックによるハング，プロセスが立ち上がらなくなるなどカーネルクラッシュに至らないケースも多く見られた．特にメモリ管理やスケジュール関数はカーネルのその他の部分よりも多くのエラーチェックがなされており，エラーを検知してフェイルストップすることも多く見られた．このようにカーネルのエラー処理が効果的に働くケースが多いことから，カーネルはエラー伝播を検知することは可能であることがわかる．

5.4 調　査　例

ここではエラー伝播の実態について，調査した典型例を紹介する．ここであげる例と同様にして，他のエラー 147 件についても調査した．

5.4.1 エラーが伝播しなかったケース

分岐ミスやエラー処理忘れを再現する BRANCH FAULT を挿入した場合の調査経過について述べる．フォールトインジェクションしたところ，表 3 の結果が得られた．このフォールトでは，分岐命令を削除 (NOP 命令に変更) するため，if 文が削除される．ターゲットとコードの場所が関連性が低いのは，コンパイルする際にインライン展開やマクロ展開されるためである．フォールトによってこの関数内では BUG() が確実に実行されるようになり，強制的に panic が呼び出され，カーネルパニックを引き起す．データ構造の変更は特にされないため，エラー伝播は発生しない．しかし，フェイラは発生する．

表 4 INIT FAULT の具体例

ターゲット	schedule+0x1ea
コードの場所	kernel/sched.c の 2668 行目
元の命令	movl $0x0,0xffffffec(%ebp)
変化後の命令	nop nop nop nop nop nop nop
元のコード	int sd_idle = 0;
変化後のコード	int sd_idle;
エラー伝播	あり

このように，致命的なエラーが発生するため，エラーが伝播する前にフェイラを引き起すケースが多く見られた．また，条件分岐系のフォールト (BRANCH，LOOP FAULT) を挿入すると，BUG_ON() などのエラーチェックコードにフォールトが挿入されるケースが多く見られ，Linux は頻繁にエラーチェックをしていることがわかる．

5.4.2 エラーが伝播したものの，フェイラを伴わず訂正されたケース

次に INIT FAULT を挿入した場合の調査経過について述べる．フォールトインジェクションしたところ，表 4 の結果が得られた．このフォールトでは，エラー伝播が発生したものの，エラーチェックにより正しい値に再設定され，エラーが訂正された．

このフォールトにより，関数の最初に sd_idle の初期化が行われず，エラーが発生する．エラーとなった変数は find_busiest_group 関数の呼び出しの際に引数として渡されたため，エラー伝播が発生したものの，関数内で引数チェックがあったため，エラーは訂正された．

このように，カーネル内でエラーチェックが適正にされているためにエラーが訂正されるというケースも見られた．しかし，ほとんどは検知するだけで訂正することはできていない．

5.4.3 エラーが伝播し，フェイラにより収束したケース

次に引数ミスを再現する INTERFACE FAULT を挿入した場合の調査経過について述べる．フォールトインジェクションしたところ，表 5 の結果が得られた．このフォールトにより，プログラムの起動をすると segmentation fault と表示され，起動に失敗するようになる．

ここでは多くの関数が呼び出されるため次々にエラーが伝播する．エラー伝播の様子を図

Vol.2011-ARC-195 No.6
Vol.2011-OS-117 No.6
2011/4/13

表 5 INTERFACE FAULT の具体例

ターゲット	load_elf_binary+0xa2c
コードの場所	fs/binfmt_elf.c の 352 行目 (load_elf_interp 内)
元の命令	mov 0x1c(%ebp),%edx
変化後の命令	nop nop nop
元のコード	retval = kernel_read(interpreter, interp_elf_ex->e_phoff, (char *)elf_phdata,size);
変化後のコード	retval = kernel_read(interpreter,&cachep->avail, (char *)elf_phdata,size);
エラー伝播	あり

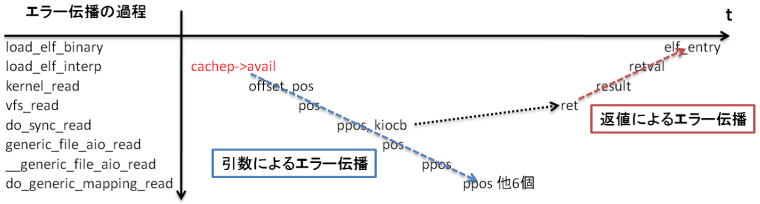

図 1 エラー伝播の様子

1 に示した．この図では下方向に関数が順次呼び出され，横方向は時間経過を表している．図 1 では，最初は関数引数を介してエラー伝播が発生するが，最終的にそれは収束し，今度は関数返値を介してエラーが伝播している．最後にフォールトの挿入された関数の呼び出し元の関数への返値がエラーとなるが，そこで返値のエラーチェックが正しくなされていたため，エラーを検知して起動しようとしたプロセスに SIGSEGV シグナルが送られ，そのプロセスは強制的に停止される．エラーはそこで適切に処理され，エラー伝播は訂正されずに収束したことになる．

このように，エラー伝播はメモリの変化の観察以外にも，カーネルソースコードで見てどのような関数を通じて発生したか，どのような変数を介して起きたかについても詳細に調査した．また，このケースのようにプロセスが立ち上がらなくなってしまうため，フェイラと判定したもののカーネルクラッシュはカーネルによって防止されたというケースも多く見られた．

6. 結　　論

本研究により，カーネルフェイラの発生過程であるエラー伝播の性質が少しずつ明らかになってきた．500 回のフォールトインジェクションを Linux カーネルに対して行ったところ，エラー発生からエラー伝播発生に至る確率は 12%であった．中でもその関数内で呼び出す関数に引数を伝わってエラーが伝播するケースが一番多い．しかし，カーネル内において頻繁にエラーチェックが行われており，フェイルストップするケースも多く見られた．エラー伝播は限定的に波及する傾向があり，特にエラー伝播はプロセス間で発生せず，関数間で発生していた．エラー伝播はフォールトの挿入された関数内で収束する確率が高く，次に呼び出される関数から順に収束する確率が高くなる傾向にあると考えられる．エラー伝播は訂正することや発生しないようにすることは難しい一方で，プロセスや関数内でアイソレーションさせることは可能であり，また検知することも可能であることがわかった．

7. 今後の課題

本研究で得られた結果は，500 回という限定的な数における調査から得られたものである．そのため，エラー伝播の正確な実態を知るためにはより多くの調査をしていくことが必要となる．現段階の方法では実質１つ１つのエラーを人間が見て確認するという原始的な方法であるため，今後はこの方法を自動化していくことが必要になると考えられる．また，調査結果で得られたいくつかの分析もより多くの実験を通してその妥当性を検討する必要がある．

本調査によりカーネルフェイラの際に発生するエラー伝播の性質が明らかになってきた．そのため，よりよいカーネルフェイラへの対策を考えることも可能となった．Linux はメモリハードウェアエラーによるエラー伝播を訂正することや発生しないようにすることは難しい一方で，カーネルの構造によってエラーがプロセスや関数内にアイソレーションされる傾向にあり，またカーネルが自身のエラーを検知することも可能であることがわかった．以上の性質を利用して，エラーがシステムに発生した際に高速にリカバリする手法を考えていくことになる．

参　考　文　献

1) Feng, W.: Making a case for efficient supercomputing, *Queue,1(7)*, pp. 54–64 (2003).
2) Patterson, D.: Recovery Oriented Computing: A New Research Agenda for a New

Vol.2011-ARC-195 No.6
Vol.2011-OS-117 No.6
2011/4/13

Century (2002). Proceedings of the 8th Intl. Symposium on High-Performance Computer Architecture.
3) Chou, A., Yang, J., Chelf, B., Hallem, S. and Engler, D.: An Empirical Study of Operating Systems Errors, *Proceedings of the 18th ACM symposium on Operating systems principles (SOSP '01)*, pp.73–88 (2001).
4) Palix, N., Thomas, G., Saha, S., Calves, C., Lawall, J. and Muller, G.: Faults in Linux: Ten Years Later, *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)* (2011).
5) Li, X., Huang, M.C., Shen, K. and Chu, L.: A realistic Evaluation of Memory Hardware Errors and Software System Susceptibility, *Proceedings of the 2010 USENIX conference on USENIX annual technical conference (USENIX ATC '10)* (2010).
6) Ziegler, J.F.: Terrestrial cosmic rays, *IBM J. of Research and Development*, Vol.40, No.1 (1996).
7) Gray, J.: A census of Tandem system availability between 1985 and 1990, *Tandem Computers Technical Report 90.1* (1990).
8) Lee, I. and Iye, R.K.: Faults, Symptoms, and Software Fault Tolerance in the Tandem GUARDIAN90 Operating Systems, *Proceedings of the 1993 International Symposium on Fault-Tolerant Computing*, pp.20–29 (1993).
9) Sullivan, M. and Chillareg, R.: Software Defects and Their Impact on System Availability - A Study of Field Failure in Operating Systems, *Proceedings of the 1991 International Symposium on Fault-Tolerant Computing* (1991).
10) ArchanaGanapathi, VijiGanapathi, D.P.: Windows XP Kernel Crash Analysis, *Proceedings of the 20th conference on Large Installation System Administration (LISA '06)*, pp.149–159 (2006).
11) Gu, W., Kalbarczyk, Z., Iyer, R.K. and Yang, Z.: Characterization of Linux Kernel Behavior under Errors, *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN '03)* (2003).
12) lun Kao, W., Iyer, R.K. and Tang, D.: FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior under Faults, *IEEE Transactions on Software Engineering*, pp.19(11): 1105–1118 (1993).
13) Barton, J.H., Czeck, E.W., Segall, Z.Z. and Siewiorek, D.P.: Fault injection experiments using FIAT, *IEEE Transactions on Computers*, pp.39(4):575–582 (1990).
14) Kanawati, G.A., Kanawati, N.A. and Abraham, J.A.: FERRARI: A Flexible Software-Based Fault and Error Injection System, *IEEE Transactions on Computers*, pp.44(2):248–260 (1995).
15) Chen, P.M., Ng, W.T., Chandra, S., Aycock, C., Rajamani, G. and Lowell, D.: The Rio File Cache: Surviving Operating System Crashes, *Proceedings of the 7th international conference on Architectural support for programming languages and operating systems (ASPLOS-VII)*, pp.74–83 (1996).
16) Swift, M.M., Bershad, B.N. and Levy, H.M.: Improving the Reliability of Commodity Operating Systems, *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pp.207–222 (2003).
17) Swift, M.M., Annamalai, M., Bershad, B.N. and Levy, H.M.: Recovering device drivers, *In Proceedings of the 2004 Symposium on Operating Systems Design and Implementation (OSDI '04)* (2004).
18) 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)*, pp.181–194 (2010).
19) Swift, M.M.: Nooks Research Group(http://http://nooks.cs.washington.edu/).