Vol.2012-ARC-200 No.8
Vol.2012-OS-121 No.8
2012/5/7

Linux 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] 河野健二[†1,†2]

OS カーネルには高い信頼性が求められている．しかし，カーネル内のバグを完全に取り除くことは困難であるため運用時に発生するエラーからリカバリする手法が求められる．本研究では始めにカーネル内で発生するエラー伝播はプロセスコンテキストに閉じるケース（プロセスローカル）とカーネル共有データに及ぶケース（カーネルグローバル）の 2 種類に分類されることを示す．エラー伝播の多くはプロセスローカルであり，カーネルグローバルであってもプロセスコンテキストは高確率でクリティカルセクション内でクラッシュすると予想される．その場合クラッシュが起きたとしてもフェイルしたプロセスをキルするとそれ以外のプロセスは正しく稼働し，デッドロックによりクラッシュ原因となったプロセス外へのエラー伝播が防がれる．分析から得られた特徴を利用してプロセスローカルエラーとカーネルグローバルエラーに対する Linux 2.6.38 の反応をフォールトインジェクションにより調査する．実験では 49 件のクラッシュについて直前に実行した命令を調査し，全てプロセスローカルエラーであることを確認した．また，445 件のプロセスキル後のエラー調査では 66.7 % はエラーが表面化せず，デッドロックが発生するケースが 24.0 % あることを確認した．

A Study on the Scope of Error Propagation in Linux

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

Operating systems are crucial for achieving high availability of computer systems. Even if applications running on an operating system are highly available, a bug inside the kernel may result in a failure of the entire software stack. The objective of this study is to gain some insight into the development of the Linux kernel more resilient against software faults. In this paper, we show two types of scope of error propagation, process-local and kernel-global. The propagation scope is process-local if the error is confined in the process context that activated it. The scope is kernel-global if the error propagates to other processes' contexts or global data structures. We believe that most errors are process-local and even if errors are kernel-global, most process contexts crashes in the middle of critical sections. Therefore, only if we revoke an erroneous context, most contexts can run correctly otherwise fail-stop with high probability even after the kernel crashes. To investigate kernel reaction against process-local and kernel-global error, we perform a kernel-level fault-injection campaign on Linux 2.6.38. In our experiment, we confirm all of the errors which caused 49 crashes are process-local. We also examine 445 errors caused after killing erroneous process and the result show that 66.7% errors are not manifested and 24.0 % errors cause dead-locks.

1. イントロダクション

OS カーネルには高い信頼性が求められている．OS に障害が発生するとその上で動作する全てのアプリケーションの障害に直結するため，カーネル障害は致命的な問題となる．

カーネルにおける障害の要因の一つはカーネル内のソフトウェアバグであり，Linux カーネルにおけるバグの調査[1)] では数年に渡って Linux はバグを完全には取り除けていないことが報告されている．SeL4[2)] ではモデルチェックを用いてバグを積極的に避けることが可能であるが，商用で使われている Linux などの OS カーネルにおいて適用することは難しい．例えば割り込みハンドラのような非同期で呼ばれる処理部分に対して正しくモデルを作ることは困難である．従って，高い信頼性を達成するためには運用時にカーネルクラッシュの対策をする必要がある．

我々は本研究において，カーネルのバグによって発生するエラー伝播の範囲をサブシステムやコンテキストの粒度で調査する．Linux カーネルは独立した多くのサブシステムにより構成されており，例えばネットワークやファイルシステムのような，コードの依存関係が低いサブシステム間ではエラーは伝播しにくいと考えられる．また，エラー伝播の範囲はプロセス毎のデータで閉じる場合とカーネル全体で共有するデータに及ぶケースの 2 種類あるといえる．例えばあるカーネルスタック内にのみエラーが閉じる場合，そのカーネルスタックを使用するコンテキスト以外のコンテキストは全て問題なく実行可能である．サブシステムやコンテキストの粒度で調査することによって Linux カーネルにおけるエラー伝播の範囲に特徴的な傾向があることを知ることができる．

本研究では Linux 2.6.38 カーネルに対してフォールトインジェクションすることで，Linux カーネルのソフトウェアフォールトに対する反応を調査する．本調査を通して，カーネルの

†1 慶應大学
Keio University
†2 JST CREST

© 2012 Information Processing Society of Japan

リカバリ手法考案への足掛かりとする．このフォールトインジェクタは合計 14 種類のソフトウェアバグを再現する．実験に用いたフォールトインジェクタは既存研究において信頼性の評価に用いられてきたフォールトや Linux におけるバグの調査[1] において調査されたバグを再現する．また，エラー伝播の範囲をサブシステム単位の粒度で分析するために，それぞれ独立したカーネルサブシステム（kernel core, ext4, fat, network sound, usb）をそれぞれ集中的に使用するワークロードを用いて調査する．調査の１つではプロセスコンテキストのフェイラ時に Linux ではプロセスキルが発生することを利用して，プロセスキル前後のワークロードを変化させることでカーネルグローバルエラーの調査をする．

調査によりエラーはプロセスローカルである確率が高いことを確認した．106 件のクラッシュのうち，フォールトの実行から 100 ステップ以内にクラッシュしているケース 49 件のエラーは全てプロセスローカルであった．また，調査によりエラーはカーネルグローバルであってもプロセスキル後はアイソレートされる確率が高く，エラーはサブシステム単位で閉じることが多いことを確認した．フェイラによって発生するプロセスキル前後でワークロードを変更しない場合はデッドロックが起こる確率が高く，ワークロードを変更した場合は何も起こらない確率が高い．

本論文の構成を以下に示す．2 章でエラー伝播のカーネルに対して及ぼす影響の範囲について説明する．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およびワークロードについて説明し，4 章で調査の内容や結果の詳細を述べる．5 章で本調査で得られた結果からプロセス単位のフェイルストップとしてのプロセスキルの可能性について議論をする．6 章で関連研究を説明し，最後に 7 章で結論と今後の課題を述べる．

2. エラー伝播

OS カーネルにおいてエラー伝播が特定の領域に限られるならば，その領域にアクセスしなければそれ以上のエラー伝播を防ぐことが可能となるためシステムの稼働を安全に継続することが可能となる．そこで本研究では，コードの依存関係が低いサブシステム間ではエラーは伝播しにくいことに着目し，サブシステムごとに発生するエラー伝播を調査する．この調査により，例えばネットワークに問題がある状況でもファイルシステムは問題なく使用できることを示す．また，プロセスキルによってプロセスコンテキストに閉じるエラーは取り除くことが可能であることと，プロセスコンテキスト外にエラーは伝播しにくい点に着目し，エラー伝播の範囲をプロセスローカルエラーとカーネルグローバルエラーに分類する．プロセスローカルエラーはプロセス毎カーネルスタックやレジスタなどのプロセスコンテキスト・一時的に確保されたヒープメモリのエラーを指し，同期をせずに読み書きするデータ構造のエラーを指す．カーネルグローバルエラーはメモリディスクリプタやファイルディスクリプタなど，全てのコンテキストが共有して使用しているデータ構造のエラーを指し，基本的にロックなどにより同期を取らなければ正しく使用できないデータのエラーを指す．

本章において例として使用するコードは全て Nooks Research Group の Web サイト[3] で利用可能なフォールトインジェクタを使用した結果発生したコード変化を示している．このフォールトインジェクタは既存研究[4] で信頼性の評価に使われており，3 章で説明するフォールトインジェクタと同様のカーネルのテキスト領域を命令単位で書き換えるものであり，逆アセンブラやデバッガを用いてコードの変化を調査した．

2.1 エラー伝播の範囲

エラーはプロセスローカルになる確率が高いと考えられる．Linux カーネルは多種多様なシステムにおいて様々な用途で使われており，多くのカーネルデベロッパーにメンテナンスされている．その結果，コードにおいてエラーチェックが頻繁かつ厳密になされており，伝播したエラーは高い確率で早期に検知することができる．また，ポインタ変数はハードウェアによってアクセス可能か常にチェックされているものの，エラー検知時にカーネルクラッシュを引き起こす．また，カーネルグローバルデータへの書き込みと比較してプロセスローカルデータへの書き込みの頻度は高いため，確率的にエラーはプロセスローカルになりやすいことが予想される．C 言語の性質上多くの計算はスタック変数を使用し，スタック外変数はスタック変数による計算が終了してからコピーされることが多いためである．また，クラッシュの 60% が フォールト地点から 10 サイクル以内で発生していることが既存研究[5] により示されており，その場合フォールトからクラッシュまでの間に実行される命令はプロセスローカルデータである確率が高く，その結果高確率でエラーがプロセスローカルとなると考えられる．よってクラッシュ時にエラープロセスコンテキストを取り消せば，高い確率でカーネルデータからエラーを取り除くことができると予想される．

図 2.1 は Linux に対してフォールトインジェクションしたときのコードを示している．BUG_ON マクロの条件式が反転するフォールトにより，エラーを誤検知してクラッシュする．このとき，クラッシュはクリティカルセクションではない位置において発生しているため，`tty_ldisc_put` や `tty_ldisc_ref_wait` による状態変化はこの時点で必ず一貫性が保たれることが保証されている．よってこのときクラッシュは発生してもデータ構造が破壊されることはないためプロセスローカルエラーである．

© 2012 Information Processing Society of Japan

```
static int tty_set_ldisc(struct tty_struct *tty, int ldisc)
{
..........
         if (!test_bit(TTY_LDISC, &tty->flags)) {
                  spin_unlock_irqrestore(&tty_ldisc_lock, flags);
                  tty_ldisc_put(ldisc);
                  ld = tty_ldisc_ref_wait(tty);
                 tty_ldisc_deref(ld);
..........
void tty_ldisc_deref(struct tty_ldisc *ld)
{
        unsigned long flags;
        BUG_ON(ld != NULL); //FAULT&CRASH
        spin_lock_irqsave(&tty_ldisc_lock, flags);
..............
```

図 1 プロセスローカルエラーとなるフォールト例（一部省略）

```
int propagate_mnt(.........)
{
        LIST_HEAD(tmp_list);                //FAULT
.........................
out:
        spin_lock(&vfsmount_lock);
        while (!list_empty(&tmp_list)) {
                child = list_entry(tmp_list.next,
                          struct vfsmount, mnt_hash);  //CRASH
                list_del_init(&child->mnt_hash);
                umount_tree(child, 0, &umount_list);
        }
        spin_unlock(&vfsmount_lock);
.........................
}
```

図 2 同期によりエラーが隔離されたコード例（一部省略）

2.2 同期機構によるエラーのアイソレーション

プロセスローカルエラーと同様に，カーネルグローバルエラーの場合でも，クラッシュを引き起こしたプロセスをキルすると，ロック等のグローバルなデータの参照は消えないためエラーを隔離することができる．例えば，ロック獲得後のクリティカルセクション内で実行中のプロセスコンテキストが例外により強制終了すると，そこで獲得されたロックは解放されず，全てのコンテキストはエラーとなっている共有データへアクセス不可能になる．その後はデッドロックにより全てのコンテキストが停止する恐れがある一方で，エラーデータの読み出しをして誤った実行をすることはないことが保証される．

図 2 は Linux に対してフォールトインジェクションしたときのコードを示している（コードの一部は省略）．フォールトにより誤ったリストの初期化をした結果，`tmp_list` のリスト構造が破壊され，エラー伝播後に `list_entry` 内でクラッシュをしている．この例では `vfsmount_lock` 変数のスピンロックを獲得して共有データへ書き込もうとするものの，フォールトによりリスト構造の初期化に失敗した結果，クリティカルセクション内でクラッシュしている．`vfsmount_lock` は `vfsmount` 構造体に関したリストのロックであるため，プロセスキル後は `mount` システムコールや `umount` システムコールを呼び出すとロック獲得待ちにより停止する．

2.3 Linux カーネルによるエラーの対処

Linux ではハンドルできない CPU 例外により検知されたエラーはプロセスキルによるプロセスコンテキストの取り消しによって対処している．コンテキスト取り消し後にコアダンプをし，システムの稼働を継続することでカーネルやカーネルモジュールのデバッグを容易にしている．これを Linux はカーネル oops と呼び，システム全体を緊急停止するカーネルパニックとは別に取り扱っている．本調査では，フェイラを起こしたプロセスをキルするために，カーネル oops 後に様々な種類のワークロードが正しく稼働するか調査し，フォールトやワークロードの種類によりカーネルの挙動がどのように変化するかを調査する．本論文では以後，カーネル oops 後の状態を「クラッシュ後」や「プロセスキル後」と呼ぶ．

3. 実　　験

本調査ではフォールトインジェクションによりソフトウェアバグを疑似的に Linux カーネルに挿入することで，Linux カーネルのソフトウェアバグによるエラー伝播の範囲を調査する．また，カーネル上で動かすワークロードによってカーネルの反応は変化するため，カーネル内の異なるサブシステムを集中的に使用する 6 種類のワークロードを使用し，プロセスキル後に発生するカーネルの機能制限を調査する．その調査により，最初に発生するプロセスキルによって 2 章で述べたエラー伝播が発生しているかを調査する．本章ではフォールトインジェクタの実装方法などの詳細や使用するワークロードの性質について述べる．

3.1 フォールトインジェクタ

本調査で使用するフォールトインジェクタはカーネルテキスト及びカーネルモジュールテキスト領域上の命令を変更することでフォールトを生成する．命令の変更位置の決定は，可能な限り C 言語レベルで発生するバグを再現するように，命令列のパターンから推測する

© 2012 Information Processing Society of Japan

表 1 使用するフォールト

フォールトの種類（略称）	詳細
delete branch (branch)	分岐を削除する
inverse branch (inverse)	分岐条件式を反転する
pointer corruption (ptr)	構造体ポインタのオフセット指定を破壊する
assignment corruption (dstsrc)	代入先や代入元のアドレス指定を破壊する
argument corruption (interface)	関数引数の参照を削除する
missing initialization (init)	初期化（定数の代入）を削除する
off by one (offbyone)	比較を 1 ずらす (e.g. > を >= に変更する)
missing alloc (alloc)	kmalloc 呼び出しを削除， NULL 代入に変更する
missing free (free)	kfree 呼び出しを削除する
bad size (size)	kmalloc 呼び出し時のサイズ指定を破壊する
string overrun (bcopy)	strcpy 等の string 命令直前のサイズ指定を破壊する
lengthn loop (loop)	ループ条件式を破壊する
var (var)	関数内でスタック上に変数を 5〜7 キロバイト確保する
delete null check (null)	関数返値の NULL チェックを削除する

ことで実現する．実際には，最初にフォールトを挿入する対象となるモジュール名（kernel core，ext4, fat など）を指定し，そのモジュールのテキスト領域内のランダムなアドレスからフォールトごとの変更対象となる命令列を検索する．実験において対象となるモジュールは kernel core, ext4, e1000, fat, snd, usbcore の 6 種類である．カーネルの稼働にとって比較的使用頻度の高いサブシステムで，またワークロードにより使用頻度が大きく変化し，かつコード領域の比較的大きなものを選択した．

表 1 は実験において用いるフォールトの種類と再現するバグである．フォールトの種類ごとに変更対象となる命令と変更の方法は異なる．フォールトは既存研究において信頼性や可用性の評価に使用されてきたフォールト[4),6),7)] や，Linux カーネルのコードにおいて見られるバグ[1)] を再現するフォールトを使用する．表 1 の合計 14 種類のフォールトを用いることで，既存研究で提案されてきた手法と同様の評価を Linux で行うことが可能となる．また，現実に観察されているカーネルコード上のバグが実際にカーネルに対して起こす影響を調査することができる．

フォールトインジェクションは全てシステムの起動フェーズが完了し，ログイン後にシェルでコマンド入力が可能になった状態でフォールトを挿入する．その後次節で説明するワークロードを起動し，カーネルの反応を観察する．

3.2 ワークロード

本研究では，UnixBench，Netperf，Aplay をワークロードとして稼働させることで，ファイルシステム，ネットワーク，サウンドデバイスで発生するエラーに対するカーネルの反応やサブシステム内エラーの相互作用を調査する．また，より近い関係にあるサブシステム間で発生するエラー伝播を調査するため，UnixBench を ext4，fat，USB デバイス上の fat の 3 種類のファイルシステム上で稼働させる．ただし， USB 上で UnixBench を稼働させる場合は USB デバイスのマウントからアンマウントまでを含めて 1 つのワークロードとする．UnixBench は複数のベンチマークから構成されており，各種ベンチマークの結果は複数回実験をした平均値を結果とする．本研究では全ての実験を 1 回ずつ稼働させるように変更し，どのサブシステム上でもおおよそ 10 分間に渡って稼働する．Netperf は TCP クライアントとして 10 秒間の通信を行い，Aplay は WAV ファイルを 10 秒間再生する．また，総合的にカーネルの機能を使用するワークロードとして，システム上で動くデーモンを全て再起動するワークロードを作成し，本論文では Restartd と呼ぶ．Restartd はおよそ 30 秒間稼働する．実験ではワークロードの実行中にフォールトが実行された場合のみを実行されたフォールトとカウントし，それ以外の例えばワークロードの実行が完了された場合はカウントしない．そのため，最大でも UnixBench で稼働する 10 分間のみ調査し，本研究では長時間システムを稼働した場合に発生するエラーは調査しない．

実験に使用するカーネル Linux 2.6.38 はカーネルコンパイル時に ext4 と USB デバイスドライバをモジュールに指定する．カーネルには自作のフォールトインジェクタを追加し，kdb のバグを修正したカーネルを使用する．このカーネルはスタックフレーム内でオーバランを検知した場合にパニックを呼び出し，カーネルテキストも書き込み保護をかけている．また，スラブ内のオーバランや二重解放を検知するデバッグオプションはオフになっている．実験の一部ではカーネルを instrument するフレームワークやクラッシュ時のコアダンプを収集する kdump を利用する．

4. エラーに対するカーネルの挙動分析

3 章で述べたフォールトインジェクタとワークロードを利用することで，2 章において分析した OS カーネルにおけるエラー伝播の特徴に着目し，発生するエラー伝播の範囲を調査する．本調査では合計で 5262 件のフォールトとワークロードの組において実験し，そのうち 613 件の実行されたフォールトに対するカーネルの反応を調査する．その反応の中で，106 件のクラッシュを分析し，プロセスキルで消滅しないエラーに対するカーネルの反応を合計 445 件調査する．

本調査の結果に対して以下の内容を調査する．

© 2012 Information Processing Society of Japan

- カーネルにおけるフォールト実行とワークロードの関係
- フォールトとワークロードの組により発生するエラーに対するカーネルの初期反応
- ワークロードの種類とプロセスキル後に残るエラーに対するカーネルの反応
- クラッシュ直前の実行命令数及びプロセスローカルエラーの割合

この分析により，プロセスコンテキストの粒度でエラー伝播を分析することが可能となる．具体的にはフェイルストップによってエラーがどの程度消滅し，どの程度アイソレートされ，どの程度のエラーが残ってしまうのかを調査するために，エラーがプロセスコンテキスト，クリティカルセクション内，それ以外，のいずれかの範囲に伝播したかをカーネルの反応から推定する．

調査環境は全て Windows7 をホストとし， VMware Workstation 8 上で VM として稼働する Linux 2.6.38 である．VM のメモリは 1GB ， CPU は 64 ビットで動作する 1 コア（ホストの CPU は 2.53GHz Core2 Extreme），ハードディスク 20GB として調査した．Linux カーネルのコンパイル設定は基本的に Fedora 15 の設定を使用し，3 章で述べた修正を加えたものを使用する．

4.1 エラー原因とワークロードの関係

始めに，フォールトがどのように挿入されたのかを分析する．バグが存在する命令の分布は必ずしも一様ではなく，またカーネル内のバグが実行されるかどうかはユーザ空間上で動くアプリケーションの要求次第である．そのため，フォールトとワークロードの組によってカーネルの反応は変わる可能性がある．まず，クラッシュやエラーの分析の前にフォールトの分析をする．この分析のため，我々はカーネルデバッガを利用し，フォールトの挿入される命令にブレークポイントを設定して各種ワークロードを稼働させたときにブレークする数を調査した．

図 3 は挿入したフォールト位置ごとの実行されるワークロードの種類の割合を示す．core, ext4, e1000 のようなカーネルの中心となるサブシステムはどのワークロードでも一定の割合で通過し，fat, snd, usbcore のようなカーネルの中心でないサブシステムは特定のワークロードで通過しやすいことがわかる．この結果と後述の結果の相関からフォールトの位置とエラー伝播の関係を推測することが可能となる．

4.2 エラーに対するカーネルの初期反応

フォールトを実行すると，カーネルはクラッシュのみではなく様々な反応をする．そしてフォールトの種類や位置・稼働するアプリケーションによってさらにカーネルの反応は異なるため，この分析によりエラーのリカバリ手法において対処する必要のあるエラーを明らかにする．本調査では，前節で調査したフォールトが実行された場合のフォールトとワークロードの組についてフォールトインジェクションし，発生するカーネルの反応を分析する．

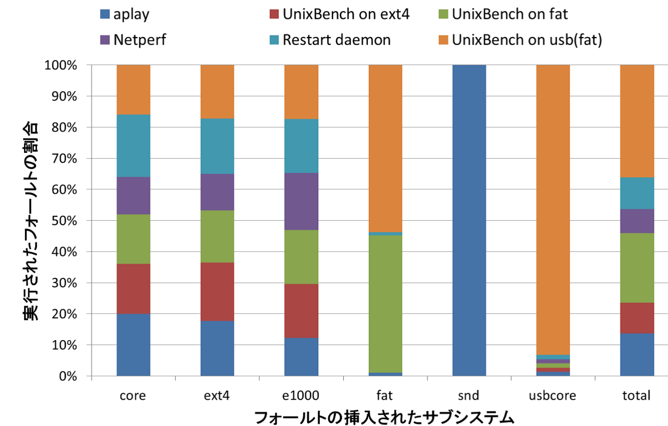

図 3 フォールト位置ごとの実行されたワークロードの割合の関係

本調査では，フォールトとワークロードの組によってカーネルの反応は変わる可能性があるため，フォールトとワークロードの組を 1 つのセットとして数える．また，カーネルの反応を完全に区別可能な種類である，panic，oops，hang，error，not manifested，terminated に分類している．panic 及び oops はカーネルメッセージから判断し，hang はフォールトによりキー入力を受け付けず，操作不可能になったケースを指す．not manifested は稼働したワークロードが最後まで正しく実行された場合を指し，途中でワークロードが停止してかつカーネルパニックや oop，ハングでないケースを error としている．terminated は VMware workstation によりシステムが停止されるケースであり，恐らくカーネルメモリのガード領域やデバイスが使用する領域で， VMware が監視するメモリがオーバランなどで破壊されると発生すると考えられる．

図 4 はフォールトの位置に対するワークロードごとのカーネルの反応を示している．ここではカーネルの反応がワークロードを起動する前に発生する場合があるため，その場合を other に分類して分析している．other のほとんどは割り込みによって発生するもので，netperf によるクラッシュが発生していない要因として，問題が発生する場合は先にネットワークに関するデーモンがクラッシュを引き起こしているためであると考えられる．Linux

ⓒ 2012 Information Processing Society of Japan

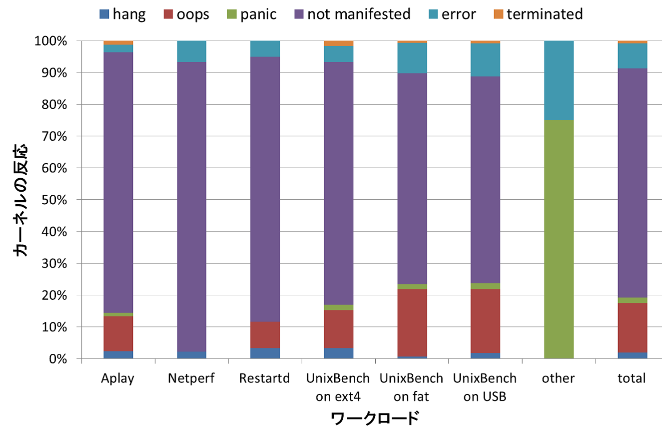

図 4 フォールトの位置と初期反応の割合の関係

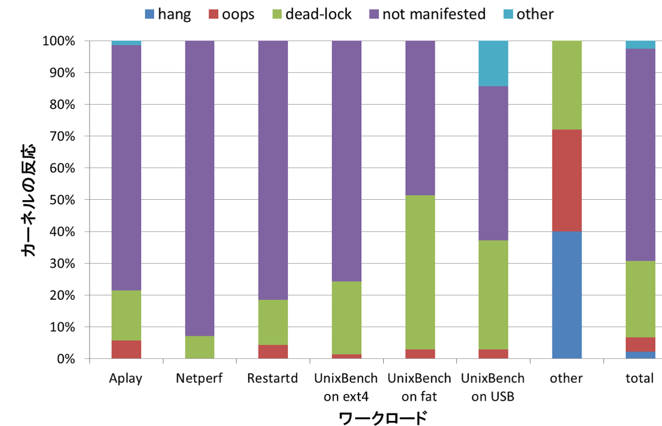

図 5 フォールトの位置と二次反応の割合の関係

カーネルのページフォルトハンドラは最初に割り込みハンドラ内であった場合にパニックを呼び出す仕組みになっているため，other においてパニックが発生しやすくなっていると考えられる．

4.3 カーネルの 2 次反応によるカーネルグローバルエラーの調査

エラー対策機構を新たに作成することを考慮する場合は必ず，カーネル内のエラーが検知された時にシステム全体を停止しないためにカーネルがより危険な状態になってしまう状況を避けなければならない．本研究ではエラー伝播がカーネルグローバルエラーであるケースを調査するために，Linux においてエラー検知時に発生するプロセスキルによってカーネルグローバルエラーのみ残ることを利用する．また，本実験に限っては挿入したフォールトも残るため，プロセスキル後に別のコンテキストが同じフォールトを実行して同じクラッシュを引き起こす場合がある．伝播したエラーがカーネルに及ぼす影響のみを抽出して分析するために，クラッシュ直後にフォールトを修正した後で各種ワークロードを稼働させる．

本調査では，4.2 においてカーネル oops を引き起こしたフォールトとワークロードの組 95 件について，フォールトインジェクション前に元の正しい命令を保存してから再度同じ実験を行い，クラッシュが起きた場合に自動的に起動するカーネルデバッガを利用して挿入されたフォールトを元の正しい命令に戻し，デバッガからカーネルに制御を戻す．その直後ワークロードを手動で停止し（多くはカーネルにより自動的に停止される），3 章で述べた各種ワークロードを稼働させカーネルの反応を調査する．また，全種類のワークロードに対するカーネルの反応を調査するまで最初のフォールトインジェクションからやり直す．

最初に発生したクラッシュ時にエラーが検知されたプロセスをキルすると，2 章で詳細に説明したように多くのエラーは消滅する一方で，ロックを獲得したままの状態でシステムが動き続けるためにデッドロックが発生する確率が高くなる．この調査では，フォールトに対するカーネルの 1 次反応で見られる反応の分類に加えて，このデッドロックによる停止を調査することで，伝播したエラーの範囲の予測をする．デッドロックによる停止の判定方法はデバッガの機能のうち，ps コマンドと同じ機能であるプロセスの状態を表示する機能を利用して，プロセスの状態が数秒に渡ってリソース待ち（ps で言うところの ”D”）になっていた場合，デッドロックによって停止していると判断する．

図 5 はフォールトの位置と二回目のワークロード稼働時の反応の割合を示している．core はカーネル oops を起こすケースが本実験では見られなかったため除外している．合計した結果を見ると，カーネルのエラー伝播に対する二次反応は 90% は何も起こらないかデッドロックであることがわかる．この結果から多くのエラーはプロセスローカルか，カーネルグローバルでもクリティカルセクション内でフェイルストップしていることがわかる．割り込んだデーモンが再度エラーとなったデータへアクセスしようとして問題を起こすケースが見られているが，数は少ない．その他の反応は USB デバイスやサウンドデバイスが認識でき

ⓒ 2012 Information Processing Society of Japan

Vol.2012-ARC-200 No.8
Vol.2012-OS-121 No.8
2012/5/7

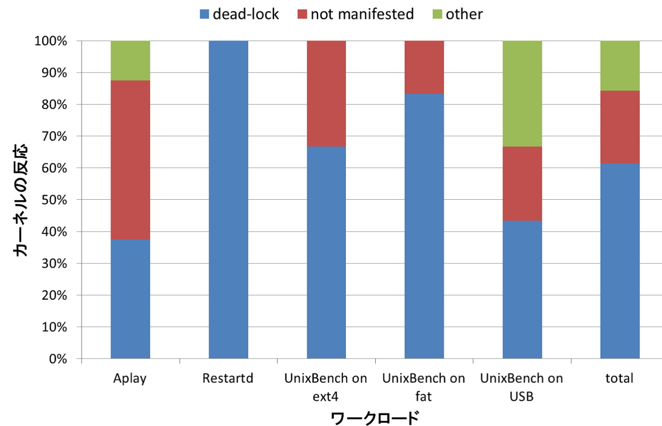

図 6　同じワークロードを再起動した場合の二次反応

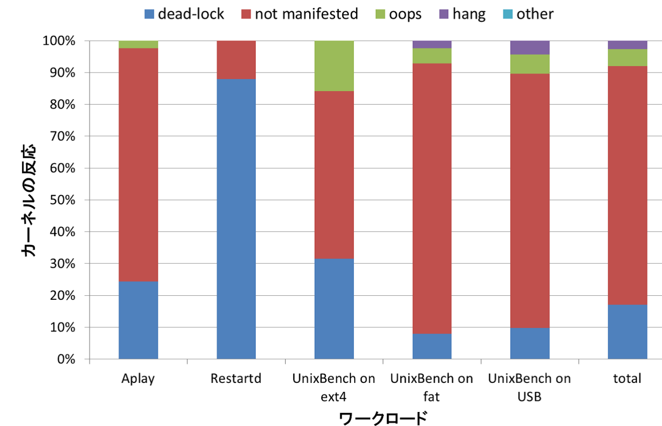

図 7　異なるワークロードを再起動した場合の二次反応

ずにワークロードが起動できないケースが見られた．二次反応ではエラーによってワークロードが停止するケースやパニックは見られず，全システムの停止をしないためにさらにシステムが危険になるような状況は見られなかった．

図 6 は二回目のワークロードのうち，1 回目と同じワークロードを稼働させた場合の反応に絞った割合を示している．このとき平均 60% はデッドロックによりワークロードを完了できずに停止していた一方で，何も起きないケースも約 20% 見られている．また，図 7 は二回目のワークロードのうち，1 回目と異なるワークロードを稼働させた場合の反応に絞った割合を示している．このとき平均 80 % は何も起こらず， 20 % がデッドロックにより停止し，一部二度目のクラッシュが発生していた．クラッシュの多くはカーネルグローバルデータ内のリストや関数ポインタによるものであった．この 2 つの分析により，エラー伝播はワークロードが使用する特定のサブシステムもしくはプロセスコンテキストに閉じていることが予想される．

4.4　実行命令履歴によるプロセスローカルエラーの調査

エラー伝播は 2 章における分析により，多くはプロセスローカルであると予想される．しかし，前節までの調査ではデータ構造による分析ではなくカーネルの反応からエラー伝播の範囲を推定しているため，not manifested であっても潜在的な問題が発生している恐れがある．そこで本節では 1 次クラッシュの直前に実行されていた命令を分析することで，プロセスローカルエラーの調査をする．クラッシュの発生しないケースについてはエラー検知が確実にできているかどうかを元の Linux カーネルでも判断する方法はないため，本調査の対象外とした．

実行した命令の調査方法として，フォールト命令から実行命令履歴とレジスタをカーネルメッセージに記録するようにカーネルを instrument するカーネルモジュールを作成する．Linux カーネルを oops 時に常にパニックを呼び出すように設定し，kdump を利用することで再起動後にクラッシュ直前のコアダンプを保存し，カーネルメッセージを抽出することで実現している．カーネルの instrument には Linux カーネルに付属しているフレームワークである kprobes を利用し，1 命令ごとにブレークして自作のハンドラに処理を移し，その中で命令とレジスタを印字する．カーネルメッセージの容量は限度があるため，本調査では 100 ステップのみ記録するようにしており，それ以上の場合は調査をしていない．またクラッシュの原因とは関係のない命令でクラッシュしている場合があり，例えばスタック溢れによって割り込みハンドラ内でクラッシュする場合，ブレークポイントを利用している本調査では調査できず，ステップ数不明として調査対象外とする．また，4.2 の結果と異なり terminated になるケースもステップ数不明としている．

調査では，4.2 において分析した oops と panic を引き起こすフォールトとワークロードの組 106 件について再度調査した．合計 106 件のクラッシュのうち 93 件について正しく

ⓒ 2012 Information Processing Society of Japan

Vol.2012-ARC-200 No.8
Vol.2012-OS-121 No.8
2012/5/7

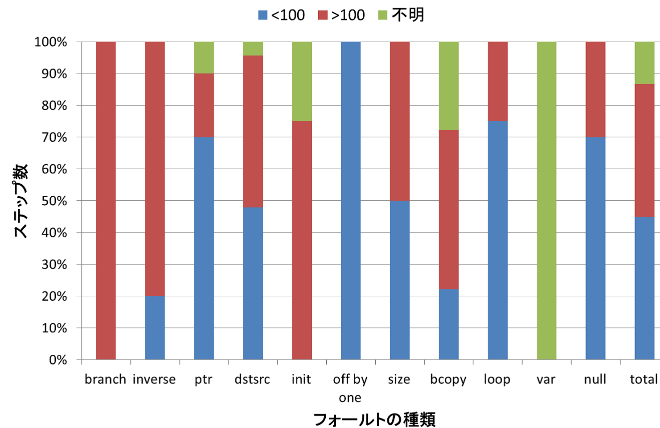

図 8 フォールトの種類ごとのクラッシュまでのステップ数

記録が可能で，そのうち 49 件が 100 ステップ以内でクラッシュを起こしていた．メモリ書き込みをするインストラクションを調べたところ，全てスタック内や一時的に確保したヒープへの書き込みでありプロセスローカルであった．プロセスローカルエラーの判別方法は，命令を手動で確認し，データフローを追跡することで判別する．その方法の詳細は，クラッシュまでの命令のうちメモリに対する書き込み命令のみを抽出し，SP や BP レジスタをベースにしたメモリ書き込みはスタック内の書き込みとして除外し，その他の残った命令に対して関数スタックフレーム内の書き込みかどうかで判断する．SP や BP 以外のレジスタをベースにしたメモリ書き込みは，カーネルコードを用いて関数内で一時的にヒープに確保したメモリに対する書き込みかどうかを確認し，そうであった場合はプロセスローカルデータであると判断し，そうでないものをカーネルグローバルデータと判断する．

図 8 はフォールトの種類ごとのクラッシュまでのステップ数が 100 ステップ以内のケースの割合を示している．結果を見ると，ポインタを破壊するフォールトやループに関するバグはステップ数が短い傾向にあることがわかる．本来より多くループを回る場合，多くはポインタの境界を越えたアクセスを引き起こすため，クラッシュが発生しやすいと考えられる．また，分岐を破壊するフォールトや初期化忘れのフォールトはステップ数が長い傾向にあり，var や bcopy のようなオーバランを引き起こすバグは本調査では分析できない傾向にある．オーバランを引き起こすバグはコアダンプを見ても何が起こっているか判断しにくく，エラーの対策でもオーバランのみ個別に対処すべきであることを示唆している．

5. プロセスキルによるエラーリカバリ

本研究で調査したエラー伝播やカーネルクラッシュはシステムに様々な問題をもたらしている．例えばシステムは保存していないアプリケーションの状態を失ってしまい，再起動の間システムに対してダウンタイムを発生させる．この問題はデスクトップ・サーバ環境に限らず，それぞれユーザに対して不利益を生じさせる一方で，カーネルクラッシュはフェイルストップの側面もあり，エラー伝播により発生するファイルシステムの破壊等のカーネルの誤作動を早期に防ぐメリットもある．カーネルクラッシュによってもたらされる安全性の保証を最低限維持しつつ，システムの可用性を向上させるためには，カーネルはエラーが伝播しているコンポーネントのみフェイルストップすることが理想である．

本研究ではエラー伝播を区切る範囲としてプロセスローカル・カーネルグローバルを用いて調査し，100 ステップ以内にクラッシュを引き起こすエラー伝播は全てプロセスローカルであることを確認した．また，カーネルグローバルエラーの調査において，多くのエラーはプロセスキル後は隠蔽されるかデッドロックを引き起こすことがわかり，プロセスキルによってさらにシステムを破壊するケースは見られなかった．この結果はカーネルクラッシュのフェイルストップとしてのメリットを，プロセスキルによるより細かい単位のフェイルストップでも維持することができる可能性があることを示唆している．

しかし，この方法はまだ問題点が多く存在する．理由は２つあり，１つはロックによって保護されないカーネルグローバルエラーの可能性を排除しきれないことと，１つはクラッシュ時に獲得しているロックの粒度によって高い可用性が達成できる場合とできない場合があるため安定しないことである．例えばあるシステムにおいてプロセスキルによってカーネルクラッシュを回避したとしても，その直後にデッドロックが発生する可能性が高く，安全にシステムの稼働を継続するためには結局システム全体の再起動が必要となってしまう．この問題に対処する方法はカーネルグローバルデータへの書き込みの正当性を常に保証することと，ロックの粒度を可能な限り細かくすることである．書き込みの保証をシンプルに達成する方法は簡単ではなく，例えばトランザクショナルメモリを用いる手法などが考えられる[8] が，課題は多い．またロックフリーなアルゴリズムも存在するため，それぞれ個別にクラッシュが起きた場合に何が発生するか検討する必要がある．以上の課題を満たした場合，ロックの粒度はシステムのメニーコア化に伴ってスケーラビリティの確保のために細かくなっていくことが予想されることを考慮すると，Linux のようなモノリシックカーネルで

© 2012 Information Processing Society of Japan

もコンポーネント単位の再起動を実現できる可能性が高くなると予想される．

6. 関 連 研 究

フォールトに対する OS カーネルの反応の調査は数多くなされており，多くの既存研究は本研究と同様に，カーネルに対して疑似的にフォールトを挿入することで調査している．Gu ら[5] は Linux カーネルに対してフォールトインジェクションをすることで，エラーが及ぼす Linux カーネルへの影響を調査している．彼らの分析は我々と同様に，多くのクラッシュが NULL ポインタ参照等のメモリ関連のエラーや，不正な命令によって発生していることを示している．また彼らはフォールト実行から多くは 10 サイクル以内に発生していることを明らかにしており，カーネルサブシステム間で発生するエラー伝播についても分析している．我々はサブシステム間やカーネルスタック間など，データの依存関係の低い領域ではエラー伝播が少ないことに注目しより細かい粒度でエラー伝播を調査している．

Chen ら[9] や Gu ら[10] は CPU アーキテクチャと OS の異なる組み合わせにおいてフォールトの影響を調査している．これらの研究は CPU アーキテクチャや OS カーネルにおいて発生するフォールトを隠蔽するためのデザインの足掛かりとなる．我々はフォールトとしてソフトウェアバグを再現するフォールトを利用して調査しているのに対し，彼らはデバイスレベルで発生する一時的なフォールトのモデルを利用して調査している．

フォールトインジェクションによるカーネルの反応の調査に対して，Linux や Windows における実際のソフトウェアバグによる影響の調査も多くなされている．Chou ら[11] や Palix ら[1] は Linux において発生しているバグについて静的解析を用いて調査している．彼らの研究の主な目的は Linux カーネル内にある特定の種類のバグの分布や生存期間を確認することである．Ganapathi ら[12] は Windows XP におけるカーネルクラッシュのダンプを収集し，分析している．彼らの研究では Windows のクラッシュはデバイスドライバが原因であることを示している．

7. 結論と今後の予定

本研究により Linux カーネルにおいて発生するエラー伝播の範囲は特徴的な傾向が見られることが示された．クラッシュ直前の命令の実行履歴をトレースすると，105 件のクラッシュのうち， 100 ステップ以内にクラッシュしているケース 49 件は全てプロセスローカルエラーであった．また，プロセスキルにより多くのエラーは消滅し，ロック変数のエラーが残るケースがあることを確認した．プロセスキル後にワークロードを稼働させてもエラーが表面化しない確率は 66.7% であり，デッドロックが発生する確率は 24.0% であった．プロセスキル後にワークロードを稼働させた 445 件の調査の中でシステムが再起動できなくなるような深刻なエラーは見られず，2 回目に稼働させる全てのワークロードは正しく最後まで動くか，動かない場合は全く動かないかデッドロックかクラッシュにより停止していた．

この結果から，エラー伝播の範囲をプロセスローカル・カーネルグローバルと区別する方法により，バグによって発生するエラーはカーネル共有データと比較してプロセスコンテキストやクリティカルセクションと強い結びつきがあることが示唆されている．今後は調査を継続することと，エラー伝播とプロセスコンテキストの相関を客観的に分析する必要があると考えられる．

参 考 文 献

1) 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).
2) Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Norrish, M., Kolanski, R., Sewell, T., Tuch, H. and Winwood, S.: seL4: Formal Veri cation of an OS Kernel, *Proceedings of the 22nd ACM symposium on Operating systems principles (SOSP '09)* (2009).
3) Swift, M.M.: Nooks Research Group(http://http://nooks.cs.washington.edu/).
4) 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)* (2003).
5) 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).
6) Zhou, F., Condit, J., Anderson, Z., Bagrak, I., Ennals, R., Harren, M., Necula, G. and Brewer, E.: SafeDrive: safe and recoverable extensions using language-based techniques, *Proceedings of the 7th symposium on Operating systems design and implementation (OSDI '06)* (2006).
7) Castro, M., Costa, M., Martin, J.-P., MarcusPeinado, P.A., Donnelly, A., Barham, P. and Black, R.: Fast Byte-Granularity Software Fault Isolation, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)* (2009).
8) Fetzer, C. and Felber, P.: Transactional Memory for Dependable Embedded Systems, *Proceedings of the 7th Workshop on Hot Topics in System Dependability (HotDep'11)* (2011).

© 2012 Information Processing Society of Japan

9) D.Chen, G. J.-S. and Mealey, B.: Error Behavior Comparison of Multiple Computing System: A Case Study Ui Linux on Pentium, Solaris on SPARC, and AIX and POWER, *Proceedings of the 14th IEEE Pacific Rim International Symposium On Dependable Computing (PRDC '08)* (2008).
10) Gu, W., Kalbarczyk, Z. and Iyer, R. K.: Error Sensitivity of the Linux kernel Executing on PowerPC G4 and Pentium 4 Processors, *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN '04)* (2004).
11) 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)* (2001).
12) Archana Ganapathi, Viji Ganapathi, D. P.: Windows XP Kernel Crash Analysis, *Proceedings of the 20th conference on Large Installation System Administration (LISA '06)* (2006).

© 2012 Information Processing Society of Japan