Vol.2016-OS-136 No.7
2016/2/29

多様な障害へ対応したカーネルレベル障害検知機能の提案と実装

岩間 響子[1,a)]　毛利 公一[2]　齋藤 彰一[1]

概要：OS の異常動作やハングアップは全てのプロセスに影響を及ぼす可能性があるため，OS にはリカバリ機能が必要である．しかし，OS の動作全体が停止してリカバリ不能となる可能性があるため，OS の動作全体が停止する前に OS の障害を検知することが必要である．本稿では OS のリカバリ機能のトリガーとなる障害検知機能の提案と実装を行った．障害検知機能は Linux カーネル内に実装し，定期的に Linux カーネルの実行状態を示すパラメータを監視することによって OS の障害を検知する．障害検知機能を備えた Linux カーネル本体と LKM，デバイスドライバに対して意図的なバグを挿入することによって評価を行い，定義した障害に対して異常を検知することを確認した．

1. はじめに

悪意あるプログラムやバグはコンピュータの異常な動作と動作の停止を発生させる場合がある．1 つのプロセスが影響を受けた場合にはそのプロセスのみが影響を受けるが，OS の障害は，コンピュータ上で動作する全てのプロセスに影響を及ぼす．全てのプロセスが影響を受けると，ユーザが行っている作業の内容は全て失われる．このため，OS は単一障害点と呼ばれ，常に正しい動作をする必要がある．

悪意あるプログラムやバグを全て防ぐことは現実的ではないため，悪意あるプログラムやバグが存在した場合に異常な動作を発見しリカバリする機能が必要である [1] [2] [3]．OS をリカバリするためには OS の動作を監視し，異常が発生した瞬間に障害を検知する必要がある．そこで，OS の動作を監視し異常を発見するための障害検知機能の提案と実装を行った．障害検知機能は Linux カーネル内に実装することで多様な障害に対応可能とした [4] [5]．障害を意図的に発生させる Fault Injection によって障害検知機能の評価を行い，障害の検知を確認した．

本稿の構成は，2 章で想定する障害について述べ，3 章で提案手法，4 章で実装について述べる．5 章で評価に用いた Fault Injection について述べ，6 章で評価，7 章で関連研究について述べた後，8 章でまとめる．

[1] 名古屋工業大学
Nagoya Institute of Technology
[2] 立命館大学
Ritsumeikan University
[a)] kyoko@mail.ssn.nitech.ac.jp

2. 障害

既存の障害検知機能の研究では，それぞれの研究において対象とする障害は異なる．そのため，本章では障害に関する他の研究の障害の定義について述べた後，本稿で想定する障害について述べる．

2.1 障害に関する研究における障害の定義

多くの研究において障害は，システムを停止させるもの，もしくはシステムの正常な動作を妨げるものと定義されている [1] [2] [3]．システムを完全に停止させなくとも，一部の機能を使用不可能とするものと著しく動作のスループットを低下させるものも一般的に障害と呼ばれる．

文献 [1] は，Linux カーネルのソースコードに発生する可能性のあるバグを，パッチファイルやドキュメントファイルのコメントから見つけている．文献 [1] は，特に障害の原因となるものとしてスピンロックを取り扱い，スピンロックの確保と解放の際に生じる可能性のあるバグを再現している．

一方で，文献 [2] と文献 [3] では障害は拡張機能で発生することが最も多いと述べ，拡張機能における障害について述べている．拡張機能とはデバイスドライバを指しており，Linux カーネル本体とは別にインストールとアンインストールが可能な Linux カーネルの拡張機能である．

どちらの障害も発生後数秒の間にシステムに影響を及ぼし，システム停止の原因とシステムの正常な動作の妨げとなる場合がある．

2.2 想定する障害について

本稿は，システムを停止，もしくは著しくシステムに影響を与えて正常な動作を妨げるカーネルレベルで発生する障害に焦点をあてる．ユーザレベルのプロセスやソフトウェアで発生する障害については対象とせず，Linux カーネルの実行中に発生する障害について述べる．以降，カーネルレベルで発生する障害を障害と呼ぶ．さらに，本稿では文献 [1]，文献 [2]，文献 [3] で特に言及された次の障害を想定する．

- 障害 1：スピンロックによる CPU の障害
- 障害 2：メモリとプロセス生成による障害
- 障害 3：デバイスドライバによる障害

障害 1 はスピンロックによって発生する CPU に関わる障害である．Linux カーネルはスピンロックを用いることで排他制御を行うことがある．スピンロックは lock と unlock の命令のペアによって実装されており，命令の欠損と誤った順序の命令は正しい排他制御の妨げとなる．スピンロックの解放を待つ間，CPU はビジーウェイティング状態となる．そのためスピンロックが永遠に解放されない場合，スピンロックの解放を待つ CPU はデッドロック状態となる．このようなデッドロック状態による CPU の障害を障害 1 とする．なお，本稿では単にスピンロックの解放を待つ状態をビジーウェイティング状態，永遠に解放される見込みのないスピンロックを待つ状態をデッドロック状態と呼ぶ．

障害 2 は，メモリとプロセス生成による障害である．メモリの確保とプロセスの生成は，大量に繰り返すことによってコンピュータのリソースを枯渇させることが可能である [6]．メモリの確保とプロセスの生成を大量に行う Loadable Kernel Module(LKM) をインストールすると，他のプログラムへのリソースの割り当てに不具合が発生する可能性がある．

障害 3 は，バグを含んだデバイスドライバをインストールすることによって生じる障害である．しかし，デバイスドライバがバグを含んでいる場合，当該デバイスを利用するたびに誤った処理が繰り返されることによって，他のプログラムに影響を与える可能性がある．

3. 提案手法

本稿では，2.2 節で定義した障害を検知可能とするカーネルレベル障害検知機能を提案する．本機能では，障害検知をユーザ空間ではなくカーネル空間だけで動作させる．Linux カーネルの実行状態をカーネル空間から監視することによってより多様な障害の検知へ対応と，ユーザ空間を介するオーバヘッドの削減を実現する．

提案の障害検知機能は，以下に示す要件を満たすことで Linux カーネルの実行状態を監視し，障害を検知する．

- 障害の検知を一定時間ごとに繰り返す．
- Linux カーネルの実行状態を示すパラメータを取得する．
- 取得したパラメータを基に障害検知を行う．

本章はこれらの要件について詳しく述べる．

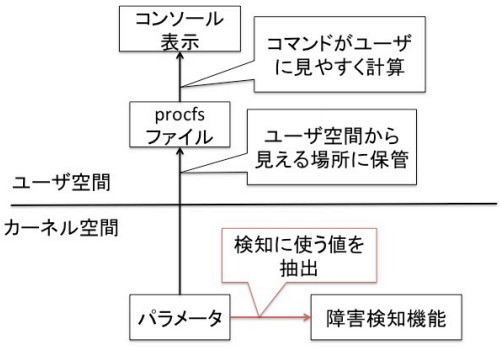

図 1 パラメータの取得方法の違い

3.1 繰り返し処理

障害検知機能は常に Linux カーネルの実行状態を監視する必要がある．そのため，障害を検知する機能は一定時間ごとに繰り返し実行する．しかし，過度な監視処理は大きなオーバヘッドとなる．障害を検知する処理は文献 [1] と文献 [2] で採用されているように，1000ms に 1 度行うこととする．文献 [1] と文献 [2] はユーザ空間で検知を行うために procfs を利用している．procfs の更新が不定期的であるため，1000ms 以下の間隔での procfs を用いた検知は正確性の低下と，オーバヘッドの増加原因となる．カーネル空間での障害検知は procfs の更新を待つ必要がないため，1000ms 秒より短い間隔での検知が可能である．本稿では，procfs を利用した既存の障害検知機構と同等以上の検知精度を低オーバーヘッドで実現するために，文献 [1] と文献 [2] と同等の 1000ms 間隔を採用する．

3.2 Linux カーネルの実行状態を示すパラメータの取得

Linux カーネルの実行状態を示すパラメータとは，CPU の使用率やメモリの使用率といった，Linux カーネルが動作する際に使用したリソースの状態とフラグの状態を指す．Linux カーネルの実行状態を示すパラメータを本稿では単にパラメータと呼ぶ．パラメータは一般にユーザ空間からコマンドや procfs を利用することによって得ることができる．本稿では，カーネルレベル障害検知機能を実現するために，パラメータをカーネル空間から取得する．ユーザ空間からパラメータを取得した場合と本稿の違いを図 1 に示す．ユーザ空間からパラメータを取得する方法では，コマンドや procfs を介する必要があるため，図 1 の赤線のようにパラメータをカーネル空間から直接取得することで

オーバヘッドの削減を実現でき，コマンドで取得できないパラメータの取得や精度の向上を期待できる．

3.3 障害の検知

障害の検知段階は，3.2 節で述べた機能によって取得したパラメータから，正常な実行状態でないものを検知する．既存研究において，障害検知機能は，異常と判断するパラメータと現在のパラメータを比較することで異常の検知を行う．正常な実行状態と異常な実行状態を区別するために，異常な実行状態を定義する．定義の詳細は 4.2 節で述べる．

4. 実装

Linux2.6.32.65 のカーネルに障害検知機能を実装した．Linux カーネルのコンパイル時に障害検知機能を選択できるように，障害検知機能はモジュールとして実装した．この障害検知モジュールは一定時間ごとに障害検知処理を繰り返し実行する．本章では，障害検知モジュールによる障害検知処理の詳細を述べる．

4.1 障害検知モジュール

障害検知モジュールは，一定時間ごとに障害検知処理を実行する．障害検知モジュールは障害によって停止してはならないため，カーネルタイマーを用いて障害検知処理を定期的に実行する．カーネルタイマーは，タイマーを起動してから変数 jiffies が事前に設定したタイムアウト値となった時に，登録したタイムアウトハンドラとして登録した関数を実行する．本実装では，タイムアウト値を 1000ms として設定した．

jiffies は，タイマー割り込みごとに加算される変数であり，タイマー割り込みが停止しない限りは動作し続けることが可能である．タイマー割り込みは Linux カーネルの制御の基本となる時間を示しており，最も優先される割り込みである．そのため障害が発生し Linux カーネルの実行状態に異常が発生した後も，タイマー割り込みが動作している間は障害検知モジュールは動作し，障害検知処理を実行することが可能である．作成したタイマーを繰り返し実行することによって，定期的な障害検知処理の実行を可能とした．

4.2 障害検知処理

障害検知処理は障害検知モジュールによって繰り返される．障害検知処理は Linux カーネルの実行状態を示すパラメータを取得し，パラメータを基に異常検知を行う．本節では障害検知処理の実装について述べる．

4.2.1 Linux カーネルの実行状態を示すパラメータの取得

実行中のカーネルの情報は，procfs を経由することでユーザ空間からいつでも参照可能であり，パラメータを取得することができる．ユーザ空間で利用できるコマンドによって得られるパラメータは，procfs の情報を基にして計算されている．しかし，procfs の情報が更新される間隔は不定期的であり，古い可能性がある．そこで，本実装では，procfs に依存せずに各パラメータをカーネル内の変数から取得する処理を実装した．

取得するパラメータは以下の 3 種類のパラメータを選出した．文献 [2] において，障害 1 と障害 2 の検知のために CPU 使用率，メモリ使用率，プロセス情報が障害の検知に有効であると述べられているためである．検知についての詳細は 4.2.2 節で述べる．

- CPU 使用率
- メモリ使用率
- プロセス情報

CPU 使用率を取得する処理は，各コアごとにカーネルコードの処理のために費やした CPU 時間，カーネルコード以外の処理に費やした CPU 時間を取得する．これらの時間は，コアごとの CPU 情報を格納している kstat_cpu 構造体を参照し取得した．取得した CPU 時間は計算によって CPU 使用率とする．計算式は vmstat コマンドのソースコードを参考とした．一例として計算式 (1) を利用した．また，CPU 使用率を取得する処理は 1000ms あたりのコンテキストスイッチの回数も取得する．

$$\text{カーネルコードの処理の CPU 使用率} = \frac{\text{カーネルコードの処理に費やした CPU 時間}}{\text{トータル CPU 時間}} \quad (1)$$

メモリ使用率を取得する処理はメモリの空き容量とページキャッシュの容量を取得する．メモリに関する情報を格納している si_meminfo 構造体を参照し取得した．

プロセス情報を取得する処理は，システム全体で動作しているプロセスの数と状態を取得する．コアごとのプロセスのランキュー情報を持つ cpu_rq 構造体を参照し，コアごとの値を足し合わせることで取得した．取得したのはブロック状態のプロセス数と割り込み不可状態のプロセス数である．割り込み不可状態のプロセス数は．CPU 使用率とメモリ使用率を求める sar コマンドでは求められないため，ps コマンドを用いる必要がある．本機能では割り込み不可状態のプロセスの数をカーネル空間から求めた．割り込み不可状態のプロセスの使用するリソースは，プロセスが終了するまで解放されず，他のプロセスの動作の妨げとなる．さらに，割り込み不可状態のプロセスはシステムが終了するまで kill されないため，常にプロセステーブルを圧迫することになる．割り込み不可状態のプロセスはデバイスドライバによる I/O 待ち状態のプロセスであるが，複数のプロセスが割り込み不可状態となることは，デバイスドライバの障害を示す．そこで，本稿はカーネル空間で，割り込み不可状態を示す uninterruptable フラグを持つプ

表 1　検知対象となるパラメータ

パラメータ	CPU			メモリ	プロセス	
	sys	usr	cs	mem	blk	flag
障害 1	✓	✓	✓			
障害 2				✓	✓	
障害 3						✓

ロセスの数を数えることで，割り込み不可状態のプロセスの数を監視した．uninterruptable フラグを持つプロセスは各コアのランキュー情報から得た．

CPU 使用率，メモリ使用率，プロセス情報はユーザ空間で sar，ps，vmstat コマンドを用いることによっても確認することができる．そのため，これらのパラメータはユーザ空間から監視することが可能である．しかし，これらのコマンドは procfs を利用しており，複数のコマンドを用いることでオーバヘッドの増加が考えられる．本稿はカーネル空間から情報を得ているため，複数のパラメータを得るオーバヘッドはコマンドを用いるものより小さいと考えられる．

4.2.2　取得したパラメータによる異常検知

検知の対象となるパラメータと 2.2 節で述べた障害の対応を表 1 に示す．CPU に関するパラメータは 3 種類あり，カーネルの処理に費やされた CPU 使用率を sys，それ以外に費やされた CPU 使用率を usr，コンテキストスイッチの数を cs とする．メモリに関するパラメータは，メモリの空き領域とページキャッシュの容量を合計したものを mem とする．メモリの空き領域とページキャッシュを合計することによって使用可能なメモリの残量を算出する．プロセスに関するパラメータは，ブロック状態のプロセスの数を blk とし，割り込み不可プロセス数のパラメータは uninterraptable フラグを持つプロセス数を示し，flag とする．取得したパラメータは直前に取得した状態を記憶して比較することで異常検知に利用した．

障害 1 及び障害 2 の検知の条件は文献 [2] の障害の調査を基にした．文献 [2] では本稿の障害 1 と障害 2 に相当する障害の評価を繰り返し，その傾向をまとめている．これらの障害検知条件は，文献 [2] で言及された傾向を基に設定した．以下，障害 1 と障害 2 の条件について述べる．

障害 1 はスピンロックに関する CPU の障害を想定した．そのため sys と usr，cs が検知に必要である．障害検知機能は，sys が 95%以上かつ usr が 5%以下であり cs が 0 である時，障害 1 を検知する．複数のコアがこの状態となった時，もしくは 1 つのコアがこの状態となって 5 秒以上が経過した時に，異常と判断して検知する．この状態の時，コアはスピンロックの未解放によるデッドロックを引き起こしていると考える．

障害 2 はメモリとプロセス生成による障害を想定した．mem と blk が検知の条件である．障害検知機能は，mem が主記憶領域全体の 3%以下となり，blk がコア数の 3 倍を越した時，障害 2 を検知する．この状態の時，fork の多発によるリソースの枯渇とランキューの圧迫によって Linux カーネル及び他のプログラムの挙動が不安定になっていると考えられる．

障害 3 はデバイスドライバによる障害を想定した．障害検知機能は flag が 1000ms あたり 10 以上増加する時に障害 3 を検出する．デバイスドライバによってスリープされたプロセスが増加することによって．プロセスの数が増加し続ける可能性が考えられる．本稿では flag が 10 以上の時に警告を行う検知機能を想定した．割り込み禁止状態のプロセスはカーネルの動作を停止させることはないが，プロセステーブルには常に 0 であることが望ましいため，本稿では 10 以上で警告を行い，ユーザに知らせることを目的とする．

5. Fault Injection

Fault Injection(FI) は，意図的に障害を発生させ，障害を再現する手法の総称である [7]．作成したプログラムのデバッグや，検知機能の評価を行うために用いられる．FI には Hardware Fault Injection(HFI) と Software Fault Injection(SFI) の 2 種類がある [8]．本章ではこれらの FI について述べた後，本稿で利用する FI について述べる．

5.1　Hardware Fault Injection

HFI はハードウェアへの障害を想定した FI である．ディスクやメモリといったハードウェアを物理的に破損，喪失させることによって障害を再現する．物理的な障害を再現することが可能で，実際に起こりうるハードウェアの障害を再現することができる．しかし，1 度障害を引き起こすごとに新しいハードウェアを必要とする．

5.2　Software Fault Injection

SFI はソフトウェアへの障害を想定した FI である．ソフトウェアに手を加えることで，ソフトウェアの実行中に発生する障害を再現する．ソフトウェアのソースコードに障害を発生させる処理を組み込んだり，実行に不具合が生じるような記述を行うことで障害を発生させる．

SFI において障害を挿入するタイミングは 2 種類あり，コンパイル時とソフトウェアの実行時である．コンパイル時に障害を挿入する方法は，障害を任意の場所で発生させることが可能である．しかし，一度組み込まれた障害は，ソフトウェアの実行中に常に実行される．そのため，ソフトウェアが起動しない可能性がある．

一方，ソフトウェアの実行中に障害を挿入する方法は，ソフトウェアが常に誤った挙動を行うわけではなく，ソフトウェアの実行中に障害の有無を調節することが可能である．例えば，Linux カーネルを正常に起動させた後に，

表 2　評価環境

CPU	Core i7-3770,3.40GHz
メモリ	16GB
OS	CentOS6
kernel	2.6.32.65

表 3　障害の種類と評価プログラム

障害 1	LKM におけるスピンロックによる障害
	Linux カーネル本体におけるスピンロックに関する障害
障害 2	LKM 内のスレッド生成による障害
障害 3	キャラクタ型デバイスドライバの障害

Linux カーネルに対して異常を与える処理を実行するコマンドを送ることが可能である．

5.3　本稿で利用する Fault Injection

本稿では，対象とする障害が SFI によって再現可能であるため，SFI の手法を用いて Linux カーネルに対して障害を発生させ，障害検知機能の評価を行う．5.1 節，5.2 節で述べたように，SFI は HFI と比較して安価に行うことができるというメリットがあるためである．

また，障害を挿入するタイミングは，Linux カーネルの実行中に挿入する方法を採用する．障害発生箇所を制限される可能性があるが，障害発生の影響で Linux カーネルが起動しない問題を回避し，起動後の障害の管理を容易にするためである．

6.　評価

本章は提案の障害検知機能の評価について述べる．評価環境を表 2 に示す．2.2 節及び 4.2 節で述べた，3 種類の障害を FI によって再現するプログラムを作成した．既知の 3 種類の障害と評価に用いたプログラムの対応を表 3 に示す．FI を行う場所はカーネル空間とし，障害によってカーネル本体，LKM，デバイスドライバのいずれかへの FI を行った．本章ではまず初めに障害検知機能のオーバヘッドについて述べる．次に，表 3 に示した 4 種類のプログラムの説明と障害検知機能による検知結果を示す．

6.1　障害検知機能のオーバヘッド

本障害検知機構のオーバヘッドを評価するために，障害が発生しない状態において UnixBench5.1.2 を用いて実行時間を計測した．計測に用いたベンチマークは，512Byte のデータのパイプ処理を繰り返すものである．この処理にかかる時間を調査することで，CPU と OS の処理性能を算出する．

障害検知機能を動作させない状態と，動作させた状態でそれぞれベンチマークを動作させ，評価を行った．評価結果を表 4 に示す．評価結果は 128MB のメモリと Solaris2.3 の OS での処理性能を 10 として正規化した数字を示している．オーバヘッドは約 0.5%であり，軽微であることを確認した．

表 4　UnixBench5.1.2 の実行結果

障害検知機能なし	6961.4
障害検知機能あり	6936.0

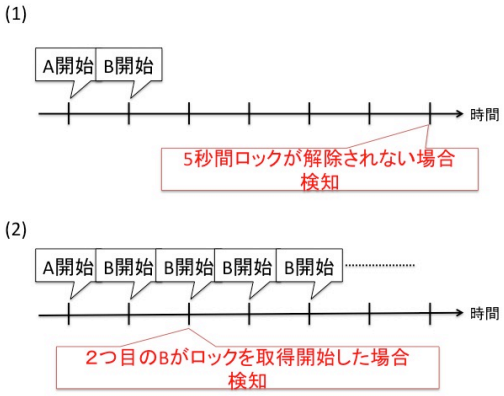

図 2　LKM 同士のスピンロックによるデッドロックの検知

6.2　LKM におけるスピンロックによる障害の検知

本節は，他の LKM と同時に動作することでデッドロックを発生させる障害を持つ LKM を作成し，評価を行った．LKM で評価を行うことで，カーネル空間の一部における障害の評価を行う．Linux カーネル本体への障害は，6.3 節で述べる．

LKM 同士は排他制御を行うためにスピンロックを用いる．1 つの LKM がスピンロックを取得した後，クリティカルセクション実行中にスピンロックを取得しようとした他の LKM は，スピンロックが取得可能となるまでビジーウェイティング状態で待機する．しかし，スピンロックを取得した LKM の行う処理が長時間にわたる場合，複数の LKM がスピンロックを取得しようとしてビジーウェイティング状態となり，デッドロック状態となってシステムは停止する可能性がある．そこで評価のためスピンロックを取得したまま長時間解放しないモジュール A とスピンロックを取得するモジュール B を作成した．評価には次の 2 通りの状況を想定し，障害検知機能による検知を行った．この時の様子を図 2 に示す．これらの障害は障害 1 の検知条件によって検知された．

(1) モジュール A が動作開始した後，1 つのモジュール B が動作を開始する．
(2) モジュール A が動作開始した後，1000ms 毎にモジュール B が動作を開始する．

(1) の時，モジュール B が動作を始めて 5000ms から 6000ms 後に障害を検知した．障害検知機能は 1 つのコアがビジーウェイティング状態となって 5000ms 以上回復し

なかった場合をデッドロックと検知するためである．検知機能のタイムアウト時間である 5000ms を除くと，障害は 1000ms 以内に検知された．

(2) の時，最初のモジュール B が動作を始めて 1000ms から 2000ms 後に障害を検知した．障害検知機能は 2 つ以上のコアがビジーウェイティング状態である場合をデッドロック状態と検知するためである．障害検知機能は 2 つ目のモジュール B が動作を開始した後に検知した．2 つ目のモジュール B が起動するまでにかかる 1000ms を除くと，障害は 1000ms 以内に検知された．検知後に全てのコアがビジーウェイティング状態となると，システムはモジュール A がスピンロックを解除するまで完全に停止した．

(1)，(2) どちらの場合においても障害は 1000ms 以内に検知可能であった．本障害検知機能は 1000ms に 1 度検知を行っていることから，障害の発生後最初の 1 サイクルでの検知が可能であったと考えられる．

6.3 Linux カーネル本体におけるスピンロックに関する障害の検知

本節は Linux カーネル本体のソースコード内に FI を挿入してスピンロックを発生させた障害について述べる．評価を行った障害はさらに 2 種類に分けられる．

- Linux カーネルの動作を完全に停止させる可能性のある障害
- Linux カーネルの動作を完全に停止させることはないが，挙動を制限する障害

これら 2 つの状況を引き起こす障害の発生方法と障害検知機能による評価結果について述べる．また，本節における FI を挿入した場所は文献 [1] を参考とした．

6.3.1 Linux カーネルの動作を完全に停止させる可能性のある障害

あるスピンロックを lock したプロセスが unlock しない場合，Linux カーネルは完全に停止する．他のプロセスが新たに当該スピンロックを lock することによりデッドロックが発生するためである．Linux カーネル内でスピンロックを使用している箇所の unlock を削除する FI によって障害を発生させた．

この障害は障害 1 の検知条件によって検知された．2 つのコアがデッドロック状態となったことを検出したため，障害によって複数のコアが 1 つのスピンロックに対して lock を行ったと考えられる．

6.3.2 Linux カーネルの動作を完全に停止させることはないが，挙動を制限する障害

本節ではスピンロックに関する障害のうち，6.3.1 節で述べた障害とは異なり，Linux カーネルを完全には停止しない障害について述べる．この障害は Linux カーネル上で動作する他のプログラムの挙動を著しく妨げる可能性がある．本節で扱う障害は次の 2 種類である．

- 同一スピンロックの lock と unlock の両方が喪失する障害．
- フラグを設定したスピンロックのフラグの解除をしない障害．

前者は，同じスピンロックを 2 度連続で用いる場合に，ふたつのロックの間の unlock と lock を消すことによって 1 つの処理とする障害である．このようなスピンロックを作成すると長く 1 つのスピンロックを使用することになるため，望ましくない．Linux カーネル内で同じスピンロックを連続で用いる箇所に対して FI を行った．

後者は，スピンロックを取得する際のフラグに関する障害である．スピンロックを取得する時，スピンロックの取得中に割り込みを禁止するフラグを設定することが可能である．このフラグを設定するスピンロックを lock した場合，プロセスは unlock する際にフラグを解除する必要がある．本節ではフラグを解除せずにスピンロックを unlock した場合，フラグが解除されないため割り込みが禁止されたままとなる障害を想定した．Linux カーネル内で割り込み禁止フラグを設定するスピンロックを用いる箇所に対して FI を行った．

本稿の障害検知機能では，2 種類の障害を検知できず，これらの障害は 6.3.1 節のような障害 1 の検知条件では検知不可能であることがわかった．障害 1 の検知条件は，CPU の利用率とコンテキストスイッチの回数を基にしており，スピンロックによるデッドロックの検知に向いている．しかし，本節の障害はどちらもスピンロックによるデッドロックは発生しない．そのため，障害 1 の検知条件では検知不可能であったと考えられる．この障害を検知するためには，文献 [1] で言及されているように，障害検知機能から信号を送信し，障害状態となっていないことを確認する必要がある．この手法の詳細は 7.1.1 で述べる．

6.4 LKM 内のスレッド生成による障害の検知

本節は LKM 内でスレッドを生成することによって，リソースとランキューを圧迫する障害を想定した評価を行った．スレッドの生成によって複製するプロセスの数が急激に増加すると，リソースとランキューが圧迫され，障害が発生する [6]．また，文献 [2] は fork によってシステムを不安定とするユーザプロセスの fork 爆弾を障害の例に挙げ，カーネル空間で同様の状況を発生させるために LKM として実行することによって評価に用いている．そこで，本稿は無制限にスレッドを複製しメモリを確保する fork 爆弾に近い LKM を作成し，インストールする FI を実行した．作成した LKM はスレッドを一度に 1000 個作成し，1 つのスレッドが 1000ms ごとに 1MB のメモリを確保する．1000 個以上のスレッドを一度に作成すると，障害検知機能が動作する前に Linux カーネルが動作を停止したため，作成するスレッドは 1000 個とした．

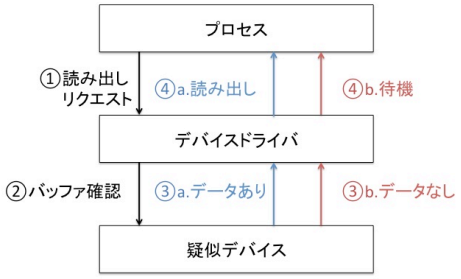

図 3 デバイスドライバの読み出し時の動作

この障害は障害 2 の検知条件によって，LKM を実行開始してから約 16000ms から 17000ms 後に検知された．作成した LKM の各スレッドによって 1000ms 毎に 1GB のメモリが消費され，評価に用いた計算機のメモリサイズは 16GB であるため，LKM を開始した 16000ms 後にメモリの空きが 3%である 512MB 以下となったと考える．さらに，メモリ不足によりメモリ確保ができず，さらにブロック状態となったスレッド数がコア数の 3 倍を超えたと考える．以上より，異常状態となった 16000ms 経過直後に検知できたと考えられる．

本稿の障害検知機能は，メモリ残量とブロック状態のスレッド数に基づく異常検知が可能であることを示した．今後は，より早期の検出を目指して，時間当たりのメモリの消費量に基づいて異常検知を行う方法を検討する．

6.5 キャラクタ型デバイスドライバの障害の検知

本節はキャラクタ型デバイスドライバに障害が発生し，プロセスが正常にデバイスを利用できない状態を想定した評価について述べる．実際のデバイスに障害を発生させるのはリスクが高いため，本稿は FI を行うためのキャラクタ型の疑似デバイスとそのためのデバイスドライバを作成した．本キャラクタ型疑似デバイスは，ユーザプロセスによるデータ書き込み機能と，そのデータの読み出し機能を提供する．また，データが書き込まれていない状態での読み出しアクセスをブロックし，当該プロセスを停止させる．このデバイスドライバに対してプロセスから読み出し要求があった場合の動作を図 3 に示す．

プロセスからデバイスに読み出し要求があると，デバイスドライバがそれを受け付け，デバイス内に書き込まれたデータの有無を確認する．読み出し可能なデータがある場合はデバイスドライバがそのデータをプロセスに返す．読み出し可能なデータがない場合，デバイスドライバはプロセスをブロックし，待機状態とする．この時，待機状態となったプロセスはデバイスドライバによって uninterruptable フラグを設定されており，デバイスドライバが解除するまで待機状態が継続する．デバイスドライバは他のプロセスから書き込みリクエストがあった時，uninterruptable フラグを解除し，待機状態となっているプロセスを実行可能状態とする．

このデバイスドライバへの FI として，書き込みリクエストがあってもプロセスの待機状態を解除しない障害を発生させた．さらに，デバイスドライバにアクセスするプロセスを無条件ループによって無制限に生成し，待機状態となるプロセスを増加させた．

この障害は障害 3 の検知条件によって検知された．検知時間は読み出し要求を行うプロセスが動作してから 1000ms 以内だった．障害検知機能が 1000ms に 1 度動作するため，1000ms 以内に検知することが可能であったと考える．

7. 関連研究

本章では関連研究について述べる．本稿に関係する研究として，障害の定義に関する研究，障害の検知に関する研究，及び検知した障害からのリカバリに関する研究が挙げられる．

7.1 障害の定義に関する研究

障害検知機能や障害のリカバリを提案する研究の中には提案システムに限らず障害についても詳しく述べたものがある．本節では特に障害に焦点をあて，詳しい研究がなされている研究について述べる．

7.1.1 Assessment and Improvement of Hang Detection in the Linux Operating System [1]

文献 [1] はロックの取得と解放に関係する障害を定義し，障害を検知する障害検知機能を提案している．文献 [1] は Linux カーネルの実行中に起こりうる障害は次の 4 つと定義している．

- ロックに対応する unlock が喪失した場合．
- ロックを unlock する順番を誤った場合．
- 同一ロックの lock と unlock の両方が喪失した場合．
- フラグを設定したロックのフラグの解除をしなかった場合．

これらの障害はソースコード中のバグによって引き起こされ，Linux カーネル内でデッドロックを発生させる障害である．

文献 [1] はこれらの障害を Linux カーネル内の任意の場所に組み込み，FI を用いて障害を発生させることで評価を行っている．文献 [1] で提案する障害検知機能は定期的に Linux カーネルのログを確認することによって Linux カーネルの実行状態が異常でないかを監視する．また，誤検知を防ぐために一定時間ごとに信号を発生させている．例えば diskI/O の値に変化がなかった場合，diskI/O が発生していない状態か diskI/O が発生しているにも関わらず障害のために変化がないか判断できない．それを判断するた

め，5 秒に 1 度 disk にアクセスすることによって diskI/O に変化を生じさせている．

本稿は文献 [1] で定義された障害を用いて 6.3 節で評価を行った．文献 [1] はロックに対する障害に焦点を当てているのに対して，本稿はメモリとプロセス作生成，デバイスドライバの障害についても述べている点で異なる．

7.1.2 What is System Hang and How to Handle it [2]

文献 [2] は Linux カーネルの実行中に起こりうる障害を 6 つに分類し，それぞれの障害が発生した時の Linux カーネルの実行状態を定義している．定義に沿う挙動を得た時にその挙動を障害発生として検知する機能を提案している．6 つの障害はデッドロックに関するものと無限ループに関するものに分けられる．

- デッドロックの例：fork の多発によって使用するリソースが膨大になり，他のプロセスが使用するリソースがなくなる．
- 無限ループ例：ロックの解放待ちによるビジーウェイティング．

文献 [2] は，これらのケースを FI を用いて引き起こすことによって，障害発生時の Linux カーネルの実行状態を明らかにしている．その際にユーザレベルコマンドである sar コマンドを用いている．sar コマンドは Linux カーネルが実行のログを分析し，時間あたりのスループットを算出するコマンドである．さらに文献 [2] は sar コマンドを用いた検知機能についても述べている．文献 [2] はユーザ空間のコマンドを検知に用いているのに対して，本稿はカーネル空間で検知を行っている点で異なる．

7.2 障害の検知に関する研究

本節では本稿と同様の障害検知機能を扱った研究とツールについて述べる．

7.2.1 No PAIN, No Gain? The Utifity of PArallel Fault INjection [3]

文献 [3] は文献 [2] で評価のために用いた障害検知機能を発展させた研究である．この研究では，light detector と heavy detector の 2 つの障害検知機能を提案している．light detector は常時動作し，ログの収集と簡単な検知を行っている．heavy detector は light detector においての簡単な検知で異常を検知した時に動作し，より詳細な検知を行う．文献 [3] はこのように障害検知機能を 2 つに分割することによって精度の高さとオーバヘッドの低さを同時に実現する障害検知機能を提案している．

文献 [3] は検知のためのログの収集にユーザ空間のコマンドである sar コマンドを用いているため，カーネル空間で処理を行っている本稿とは異なる．

7.2.2 watch dog timer

watch dog timer は Linux カーネルに搭載されている障害検知機能のひとつである．watch dog timer はハードウェアタイマーを利用しており，障害によって一定時間 CPU の反応がなかった場合に割り込みを発生させる．組み込みシステムに用いられることが多く，デフォルトで 60 秒で異常の発生を知らせる．タイムアウト時間が 60 秒であり長い点と，CPU のみを監視するシステムであるため他の異常には対応していない点で，本稿と異なる．

7.3 障害のリカバリを行う研究

信頼性の高い実行基盤を開発する研究 [9] では，障害の対処とリカバリについて述べている．同時実行基盤 Orthros [10] は，障害に強い OS を提案している．Orthros は常に activeOS と backupOS を動作させており，ユーザは常に activeOS を利用して作業を行う．activeOS に障害が発生した時，activeOS で実行していた作業を全て backupOS に引き継いで activeOS を終了させ，backupOS を activeOS として実行させる．

Orthros は activeOS に発生した障害を検知する必要があるが，検知の方法は backupOS からの Inter Processor Interrupt のみであり，多様な障害を検知することはできない．本稿の障害検知機能は Linux カーネルの実行状態を常にチェックすることで，障害の場所に関係なく多様な障害を検知することが可能である．このように，本稿は Orthros のような障害の場所を特定する必要のないリカバリシステムの障害検知機能として有効である．

8. まとめ

OS の障害発生から短時間でリカバリを行うため，カーネルレベル障害検知機能を提案した．カーネルレベル障害検知機能はカーネル空間で動作するため，ユーザ空間で動作させる場合に比べて，多様な障害に対応することとオーバヘッドの削減が可能である．本カーネルレベル障害検知機能は，1000ms に 1 度 Linux カーネルの実行状態を監視することによって異常発生を検知する．評価では障害の発生から 1000ms 以内に定義した異常状態を検知することが可能であることがわかった．定義した障害によって Linux カーネルが完全に動作を停止する前に障害を検知することが可能であった．また，プロセスのフラグを監視することによって，ユーザ空間では検知できなかった障害を検知可能となった．障害検知機能に発生するオーバヘッドは 0.5%程度であり，提案の障害検知機能が軽量であることがわかった．

参考文献

[1] Cotroneo, D., Natella, R. and Russo, S.: Assessment and Improvement of Hang Detection in the Linux Operating System, *28th IEEE Symposium on Reliable Distributed Systems (SRDS 2009), Niagara Falls, New York, USA, September 27-30, 2009*, pp. 288–294 (2009).

[2] Zhu, Y., Li, Y., Xue, J., Tan, T., Shi, J., Shen, Y. and Ma, C.: What Is System Hang and How to Handle It, *Proceedings of the 2012 IEEE 23rd International Symposium on Software Reliability Engineering*, ISSRE '12, pp. 141–150 (2012).

[3] Winter, S., Schwahn, O., Natella, R., Suri, N. and Cotroneo, D.: No PAIN, No Gain?: The Utility of PArallel Fault INjections, *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pp. 494–505 (2015).

[4] Bovenzi, A., Cinque, M., Cotroneo, D., Natella, R. and Carrozza, G.: OS-Level Hang Detection in Complex Software Systems, *Int. J. Crit. Comput.-Based Syst.*, Vol. 2, No. 3/4, pp. 352–377 (2011).

[5] Z, E., Cao, P., Z, K. and R.K, I.: Reliability and Security Monitoring of Virtual Machines Using Hardware Architectural Invariants, *IEEE/IFIP International Conference on Deprndable Systems and Networks* (2014).

[6] 中川　岳，川田裕貴，追川修一：プロセスのリソース隔離による Fork 爆弾攻撃の防止手法，*ComSys2015* (2015).

[7] Duraes, J. and Madeira, H.: Emulation of Software Faults: A Field Data Study and a Practical Approach, *IEEE Transactions on Software Engineering* (2006).

[8] Natella, R., Cotroneo, D., Duraes, J. and Madeira, H. S.: On fault representativeness of software fault injection, *Software Engineering, IEEE Transactions on* (2013).

[9] Wang, L., Z, K., Gu, W. and R.K, I.: Reliability MicroKernel: Providing Application-Aware Reliability in the OS, *Reliability, IEEE Transactions on (Volume:56 , Issue: 4)* (2007).

[10] Yoshida, K., Saito, S., Mouri, K. and Matsuo, H.: Orthros: A High-Reliability Operating System with Transmigration of Processes, *The 19th IEEE Pacific Rim International Symposium on Dependable Computing* (2013).