Vol.2013-OS-124 No.6
2013/2/28

仮想マシンモニタを用いた割込み処理のデバッグ手法

宮原俊介[1,a)]　吉村剛[1]　山田浩史[2,3]　河野健二[1,3]

概要：オペレーティングシステム (OS) の複雑化，多機能化に伴い，OS 内部に存在するバグが増え続けている．OS のデバッグが困難である要因のひとつに割込みによる非同期処理がある．割込み処理はその非同期性から再現性が低く，バグレポートがあっても同一の状況を再現したり，あるいはコード修正後の動作検証が極めて難しい．静的解析やモデル検査といった手法を適用しようとしても，扱うべき状態数が爆発しやすく現実的にはかなりの困難が伴う．本研究では，割込みのタイミングを意図的に制御可能にすることによりデバッグを支援する手法を提案する．提案手法では仮想マシンモニタを利用し割込みのタイミングを意図的に制御する．仮想マシンモニタの，割込みをエミュレートするコードに変更を加えることで指定したコード位置に任意の割込みを発生させることができるようにしている．この手法を用いることで，割込みの発生を故意に起こし，開発段階での割込み処理のデバッグを支援することができる．本手法の有用性を示すために，Linux に実在した割込み依存のバグを２つ再現し，提案手法により検出できることを確認した．

キーワード：割込み，仮想化，デバッグ，非決定的バグ

1. はじめに

オペレーティングシステム (OS) の複雑化，多機能化に伴い，OS 内部に存在するバグが増え続けている [1]．バグを取り除いても，新機能を追加することによってバグが追加されてしまうため，今後もバグが増え続けてしまうことが指摘されている．

OS は開発段階でバグを排除する必要がある．なぜならば，OS がバグのあるコードを実行してしまうと，フリーズやクラッシュなどにつながりカーネル上で稼働する全てのアプリケーションが停止してしまう．アプリケーションは OS の提供するメモリや I/O などを土台として動くため，OS 上で動かしているアプリケーションもたとえどんなに信頼性が高いものを使っていたとしても停止してしまう．アプリケーション停止は，サービスの停止につながり，経済的損失を発生させる恐れがある．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 [2]．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 [3]．

[1] 慶応義塾大学
Keio Uniersity
[2] 東京農工大学
Tokyo university of Agriculture and Technology
[3] JST CREST
[a)] shun@sslab.ics.keio.ac.jp

OS のデバッグが困難である要因のひとつに割込みによる非同期処理がある．割込み処理は非同期的に発生するため再現性が低く，バグレポートがあっても同一の状況を再現したり，あるいはコード修正後の動作検証が極めて難しい．例えば，バグレポートを元にバグを再現しようとしても，割込みがレポートと同じタイミングで発生することはまれである．静的解析やモデル検査といった手法を割込み処理部分に適用しようとしても，扱うべき状態数が爆発的に増加したり，コーディングの制約が強いため現実的には適用が難しい．例えば，SPIN[4] ではオートマトンを用いて状態空間を網羅的に探索することにより，与えられたコードが妥当であるかどうかを判定する．この手法では，様々な入力に応じた全ての状態を探索する必要があるため，割込み処理を含み，コード量が膨大な ＯＳ カーネルに対しては，考えるべき入力が多くなりすぎてしまうため，適用が困難になる．マイクロカーネルが対象ではあるものの seL4 [5] のようにカーネルの形式的検証を行う試みもなされている．しかし，文献 [5] によれば状態数が増えるのを防ぐために割込み等は考慮しておらず，Linux のようなモノリシックカーネルに対し，割込みまでを考慮した形式的検証が可能になるには形式的検証の進歩を待たねばならない．

本研究は割込みのタイミングを意図的に制御可能にすることによりデバッグを支援する手法を提案する．提案手法

```
void sample_code(){
read(shared_data)
....
write(shared_data)
}
```

```
void sample_handler(){
read(shared_data);
...
write(shared_data);
}
```

図 1 割込みを禁止すべき例

では仮想マシンモニタを利用し割込みのタイミングを意図的に制御する．ゲスト OS の割込みを発生させたいところで trap を起こし，操作が ゲスト OS に移る際に割込みを挿入することで，任意のタイミングで割込みを起こすことができる．また，割込みをエミュレートするコードに変更を加えることで任意の割込みを発生させることができるようにしている．この手法を用いることで，割込みの発生を故意に起こし，テストを行いたい箇所に任意の割込みを挿入することが可能となる．

提案手法をオープンソースの仮想マシンモニタ Xen 4.1.0 [6] に実装する．また，Intel VT [7] のサポートによる完全仮想化環境を利用する．検査対象の OS として， Linux 2.6.23.1 カーネル，Linux 2.6.38 カーネルを用いる．

本手法の有用性を示すために，Linux に実在した割込みのタイミング依存のバグを二つ再現し，提案手法により原因となる割込みを挿入し，バグを検出できることを確認した．確認したバグは，TCP におけるパケットを受け取るタイミングによってカーネルがクラッシュしてしまうというバグと，DCCP におけるパケットを受け取るタイミング依存でカーネル oops が発生してしまうバグである．

本論文の構成を以下に示す．２章では割込みの仕組みを説明し，割込みを適切に処理できていないバグを紹介する．３章では提案手法について説明する．４章では提案手法の実装について説明する．５章では 挿入したバグに対する提案手法の有用性の調査とその結果について説明する．６章でデバッグに対する関連研究を紹介する．７章でまとめと今後の課題を述べる．

2. 背景

2.1 割込み

割込み処理を正しくコーディングすることは難しい．割込みは非同期で発生するため，どのタイミングでどの割込みが発生してもカーネルデータの整合性が保たれていなければならない．まず，割込みは適切に禁止/許可しなければならない．割込みが発生することによって処理結果に影響を与える，クリティカルな区間があるためである．例えば，カーネルのコードと割込みハンドラが同じデータ構造にアクセスする必要がある場合，カーネルのコードにおいてデータ構造にアクセスする前に割込みを禁止しておく必要がある．

図 1 のような場合， 共有データである shared_data を sample_code で読み込んだ後, 割込みが来て sample_handler が呼ばれてしまうと，共有データの整合性が保証されない．割込みハンドラで shared_data を書き換えられた後，処理が smaple_code に戻り，shared_data がハンドラの変更を更新せず上書きされてしまう．割込みハンドラが読み書きする共有データを他のコードが読み出し，書き込みを行う間は，割込みを禁止しなければならない．

```
if (sk->sk_state == DCCP_LISTEN) {
    if (dh->dccph_type == DCCP_PKT_REQUEST) {
        .....
    return1;
    }
    if (sk->sk_state != DCCP_REQUESTING && sk->sk_state !=
DCCP_RESPOND) {
        .....
    return1;
    }
    if (dh->dccph_type == DCCP_PKT_RESET) {
        dccp_rcv_reset(sk, skb);
    return 0;
    }
....
}
```

図 2 データを適切に対処できない例

また，割込みを受け取る際，どのようなデータを受け取っても適切に処理しなければならない．たとえば，同じ割込みでも受け取るデータには様々なものが存在する．そのデータがどのようなものでもデータ構造を壊したり，NULL ポインタ参照を起こしたりしてはならない．2.2.5 で紹介するバグのコード例を**図 2** に示す．この例では，コネクションが閉じた状態にもかかわらず，RESET パケットを受け取ると，コネクションを閉じようとしてしまい，NULL ポインタを参照してしまう．この場合，dccp_rcv_reset を呼び出す前に，コネクションが閉じていた場合には分岐し，呼び出さないようにしなければならない．

割込み処理の正しいコーディングと同様に割込み処理のデバッグも難しい．割込み処理はその非同期性から再現性が低いため，動的な検証が難しい．例え **図 3** のように バグの再現手順や，Linux の コールトレースを含むようなバグレポートがあっても，割込みがレポートと同じタイミングで発生しないため，バグの再現が難しい．

また，割込みはいつでも発生しうるため静的に検証を行うことも難しい．例えば，モデル検査を行おうとすると，起こりうる状態をオートマトンとして認識しなければならない．起こりうる状態は，入力によって作用されるが，割込みはいつでも発生しうるため，あらゆる場面で割込みが発生した際の状態を考えなければならない，さらに，割り込み処理は．割込みの種類，受け取るデータにより処理が分岐してしまう．それら全ての処理，タイミングを考慮して状態を考えようとすると状態数が処理しきれないほど多くなる恐れがある．

2.2 割込みを適切に対処できていない例

linux のバグ修正記録をまとめたサイト [8] から，実際に

図 3 バグレポートの例

割込みを適切に処理できていないバグの例をいくつか紹介する．

2.2.1 共有割込みの競合状態によるバグ

共有割り込みが呼ばれるとき，競合状態が引き起こされる可能性がある．共有割り込みとは，ある割込み番号を同じ種類の複数デバイス間で共有した割り込みである．割込みが生じると、可能性がある全てのデバイスの割り込みハンドラを次々と呼び出す．そのため，共有割込みの初期化を行っている際は，同じ IRQ の割込みを禁止しておく必要がある．

典型例として，USB ドライバのバグがあげられる[*1]．USB のデバイスドライバの一つである amd5536udc が初期化されるとき，request_irq という関数を利用して割り込みハンドラに登録される．ここで，共有割り込みが呼ばれると，同じ IRQ の他のデバイスでも request_irq を呼ぶことが可能であるため，dev 構造体が完全に初期化されないことがある．なかでも lock，regs メンバが初期化されない瞬間があり，そのとき udc_irq 関数が呼び出されると，障害を発生することがある．この例は，割込みを適切なタイミングで禁止できていない例である．amd5536udc の初期化の際には，共有割込みは禁止されていなければならない．

2.2.2 割込みを不適切に許可してしまうバグ

割込みは，前に述べたように適切に禁止されるべきである．様々な関数で割込みの禁止，許可を行うが，その際，事前に禁止している割込みを許可してしまってはいけない．

典型例として，PCI イーサネットのドライバのバグがあげられる．その一つである 8139cp の cp_start_xmit という送信ルーチンを呼び出す関数において，spin_lock_irq という全ての割込みを禁止する命令を呼び出した後，spin_unlock_irq という全ての割込みを許可してしまう関数を呼び出している[*2]．このような呼び出し方をしてしまうと，別の関数で禁止されていた割込みも全て許可されてしまう．たとえば，cp_start_xmit を呼び出す前にタイマ割込みを禁止していても，ここで許可されてしまうと，適切に時間の管理ができなくなってしまう可能性がある．このバグは，検証の際に本来禁止されているはずの割込みが許可され，そこに割込みが発生しないと発現しない．この例では，spin_lock_irq，spin_unlock_irq ではなく，それぞれ spin_lock_irqsave，spin_unlock_irqrestore という関数を用いることで，元々禁止されている割込みを再び禁止することができる．

2.2.3 割込みのタイミング依存のバグ

割込みはどのタイミングで発生しても，障害を生じることなく処理しなければならない．しかし，適切に割込みを禁止していないと，障害が発生してしまうタイミングがある．

例の一つとして，Internet Control Message Protocol (ICMP) のハンドラがあげられる．あるタイミングでパケットを受け取ってしまうと，カーネルパニックが発生するというバグが報告されている[*3]．icmp_sk_exit が，ICMP のデータを削除した後に割込みが発生すると，icmp_repl() は，ICMP ソケットを必要とするためにカーネルパニックが発生してしまう．このタイミングでは，割込みを禁止している必要がある．

2.2.4 割込みデータを適切に処理できていないバグ

割込みには，様々なデータを伴う．例えば，ネットワーク割込みなどはプロトコルの種類，パケットの種類，シークエンス番号，実データなど様々なデータ領域を持つ．どの領域がどの値であっても障害が発生しないように処理する必要がある．

例の一つとして，Datagram Congestion Control Protocol (DCCP) のハンドラがあげられる．コネクションを閉じる際，CLOSE パケットを受け取り，RESET パケットを送る前に RESET パケットを受け取ってしまうとコネクションが閉じているにもかかわらず，さらにコネクションを閉じようとしてしまうため，NULL ポインタ参照が発生してしまうというバグが報告されている[*4]．本来は，サーバ側が CLOSE パケットを受け取った後，RESET パケットを送信するが，RESET パケットを受け取ってしまうと，サーバがクラッシュする可能性がある．どのようなデータであっても適切に対処できないといけない．RESET パケットを受け取った際に，コネクションを閉じようとする関数を呼び出す前に処理を分岐しておく必要がある．

[*1] git において id c5deb832d7a3f9618b09e6eeaa91a1a845c90c65 で紹介されている
[*2] id 553af56775b3f23bf64f87090ab81a62bef2837b で報告されている
[*3] id b9f75f45a6b46a0ab4eb0857d437a0845871f314 で報告されている
[*4] id 720dc34bbbe9493c7bd48b2243058b4e447a929d で報告されている

また，Transmission Control Protocol (TCP) のハンドラにおいてもバグが報告されている．TCP では，コネクション確立の際，サーバは ACK パケットを受け取る．その際，シークエンス番号が異なると，tcp_v4_md5_do_lookup という関数が呼ばれる際，引数として渡すポインタが NULL になってしまい，カーネルパニックが起きてしまうバグが報告されている[*5]．tcp_v4_md5_do_lookup を呼び出す前にシークエンス番号が，想定しているものと一致するかチェックを行い，異なっていれば呼び出さないようにしなければならない．

3. 提案手法

3.1 概要

本研究では，割込みのタイミングを意図的に制御可能にすることによりデバッグを支援する手法を提案する．提案手法では，使用者が割込みを起こしたいコード位置と割込みの種類を指定する．そのコード位置に任意の割込みを発生させる．この手法を用いることで，任意のタイミング，任意の種類の割込みを意図的に起こし，開発段階での割込み処理のデバッグが可能となる．

3.2 割込み挿入の方法

提案手法を実現するために，仮想マシンモニタ（VMM）による仮想化環境を用いる．また，Intel VT による完全仮想化環境で行う．準仮想化を行うと，ゲスト OS への大きな変更が必要になってしまうため，割込みの処理方法が変わってしまう可能性がある．そこでゲスト OS を変更せずに仮想化できる完全仮想化を用いる．仮想化環境では，仮想マシン（VM）へのデバイスのイベントなどを VMM がエミュレートする．そこで，VMM を利用することで意図的にゲスト OS に割込みを送信することができる．VMM には，ベアメタル型のものを利用する．

提案手法における割込み処理の流れを図 4 に示す．検査対象の OS をゲスト OS として動作させ，割込みを挿入したい箇所で，割込みの種類，データを指定して trap を起こす．そこで， VMM からホスト OS にデータを準備するリクエストを渡し，ホスト OS で割込みデータを作成する．Intel VT では，ゲスト OS の CPU の状態を VMCS というデータ領域で管理している．データ作成が完了すると，ホスト OS は VMM に通知し， VMM が VMCS を書き換える．書き換え完了し，処理がゲスト OS に移るとき， データ領域を参照して割込みが挿入される．これにより，指定したコード位置に割込みを挿入することができる．

割込み位置の指定方法は，ゲスト OS の，割込みを起こしたい位置に trap を起こすハイパーコール呼び出しを追加する．ゲスト OS を実行し，このコードを実行すると，

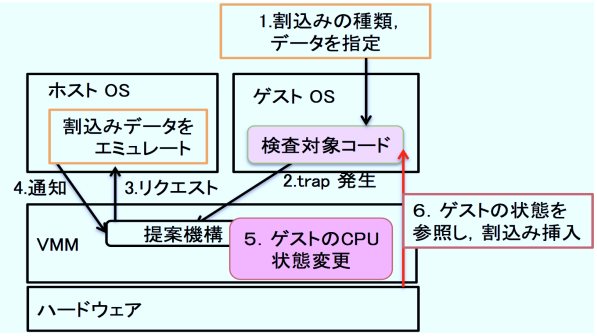

図 4 提案手法における割込み処理の流れ

操作が VMM に移る．VMM でゲスト OS に割込みを起こす準備が完了した後，操作をゲスト OS に戻す．処理が VMM に移っている間ゲスト OS は動作しないので，元のコード位置に割込みが挿入されることになる．

割込みのエミュレートは，ゲスト OS に対応する VMCS を書き換えることによって行う．ゲスト OS から操作が VMM に移った際に VMCS を書き換えることで，操作がゲスト OS に移った際に指定した割込みを挿入することができる．

また，ホスト OS において VMM から指定された，割込みのデータを作成する．仮想化環境では，ゲスト OS が割込みを直接処理するのではなく，ホスト OS が一度割込みのデータを処理し，ゲスト OS のデバイスドライバに適したデータに変換してゲスト OS にデータを渡す．割込みが発生すると，ホスト OS が代わりにデータを読み込み，ゲスト OS 用にデータを変換し，保持しておく．ゲスト OS は，仮想割り込みを受け取ると，ホスト OS にデータを読みに行くことになる．

3.3 適用方法

この手法を用いることで，Linux カーネルが異常を検知した際に表示されるエラーメッセージである Oops メッセージに報告されている，割込みが原因のバグを再現することができる．メッセージには，バグが起こったときの操作手順，問題が起こっている関数，コールトレースなどが書かれているため，問題となる割込みが発生したタイミング，種類がわかる．

まず，バグレポートより，割込み再現の操作手順やバグが発現した原因，割込みの種類などを調べる．次に実際のコードを見ることで，具体的にどのようなデータ，タイミングで割込みが発生すればバグが再現できるのかを調べる．メッセージと同じタイミングで割込みが起こるように，ゲスト OS にハイパーコールを挿入する．また，ホスト OS で，メッセージと同じ種類のデータをエミュレートする．

4. 実装

提案手法を，オープンソースの仮想マシンモニタ Xen

[*5] id 6edafaaf6f5e70ef1e620ff01bd6bacebe1e0718 で報告されている

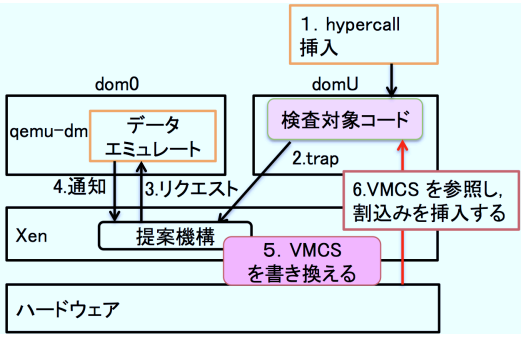

図 5 実装図

4.1.0 に実装した．Xen ではホスト OS を dom0，ゲスト OS を domU と呼ぶ．dom0 として，Linux 2.6.32.41 を使用する．

実装イメージを図 5 に示す．Xen は，domU で割込み挿入を指示する hypercall により trap が発生すると，dom0 にリクエストを投げ，データを作成する．データ作成完了後，Xen で VMCS を書き換える．Xen では，割込みデータを，dom0 内の qemu device model (qemu-dm) [9] が処理している．そこで，qemu-dm に変更を加え，データを準備する機構を作成する．

4.1 割込み挿入位置の指定

ゲスト OS では，割込みを起こしたい位置に，ハイパーコールを挿入する．ハイパーコールでは VMCALL を利用して VM exit を発生させ，制御を VMM に移す．また，VMCALL はレジスタの中身を引数として渡すことができ，割込みの種類などを指定する．そこで，キーボード割込み，ネット割込みなどの割込みの種類を指定する．また，どのようなデータを送信するかもここで指定しておく．引数に応じて処理を分岐し，必要なデータに応じた関数を呼び出すことができる．

4.2 割込みの挿入

割り込みの挿入は，Xen で VMCS を書き換えることにより行う．VMCS の VM-entry control field における，VM-Entry Controls for Event Injection という領域を書き換えることで，VM entry の際に割込みを仮想的に挿入することができる．VM-Entry Controls for Event Injection とは，VM Entry 時に仮想マシン側で割り込み，例外などを仮想的に発生させるために用いられる領域である．このなかでも，VM-entry interruption-information field を書き換えることで仮想的に割り込みを挿入することができる．挿入する Event の種類，ベクタ番号などを指示することで，指定した割り込みを挿入できる．例えば，ネット割込みを挿入する際は，イベントの種類を X86_EVENTTYPE_EXT_INTR，ベクタ番号に 0x49 を指定する．

また，dom0 でデータを準備する．dom0 は，データを準備するリクエストを受け取ると qemu-dm 上のデータをエミュレートする関数を呼び出す．データとは，キーボード割込みでいうキーコードや，ネット割込みにおけるパケットデータなどである．qemu-dm は qemu の一部であり，CPU 命令のエミューレーションや，周辺ハードウェアなどのエミュレーションを行う．完全仮想化では，qemu のデバイスエミュレーション機能を qemu-dm で実現しており，domU からは擬似的なハードウェアを見ることができる．例えば，ネット割込みを送るときには，qemu-dm 上のネット割込みのデータを domU のドライバ用に変更する関数を呼び出す．この際，挿入したい割込みのデータに合わせて関数に変更を加えておく．関数の処理が完了すると，Xen に通知する．Xen は，通知を受けると VMCS を書き換え操作を domU に移す．操作が移る際，CPU は VMCS を参照し，仮想的に割込みが発生する．

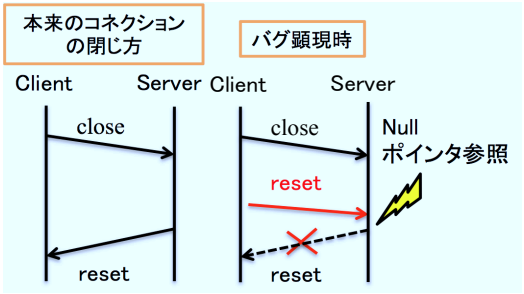

図 6 DCCP におけるバグ

5. バグの再現

5.1 目的

提案手法を用いて割り込み処理が原因のバグのデバッグを行えることを示すため，提案手法を用いて実際のバグを再現できることを確認した．

Linux の公式サイトより，割込みが原因で発現するバグを含むカーネルを domU として動作させる．その domU に対して，提案手法により原因となる割込みを挿入し，バグを発現することができるかどうかを確認する．

5.2 再現したバグ

5.2.1 DCCP に関するバグ

2.2.4 節で紹介した，DCCP に関するバグを再現する．このバグを再現するために，バグが報告されている Linux 2.6.38 を domU として使用する．

まず，図 3 に示すバグレポートにより，**図 6** のようにサーバが CLOSE パケットを受け取った後，RESET パケットを受け取ると，バグが再現されることがわかる．コードを調査することにより， dccp_ipv4_do_rcv で CLOSE パケットを受け取った後，RESET パケットを送信する関数を呼び出す前に RESET パケットを持った割込みを発生させることでバグが再現することがわかる．

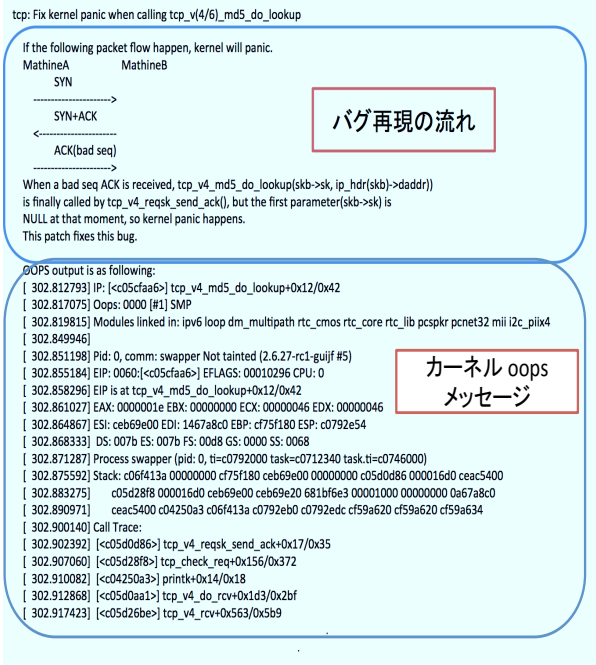

図 7 TCP のバグレポート

domU で DCCP サーバを動かし，サーバが CLOSE パケットを受け取った後にハイパーコールを挿入し，操作を VMM に移す．この際，DCCP の RESET パケットを持ったネット割込みを発生させるよう指定する．qemu-dm で，DCCP のパケットをエミュレートする関数に変更を加えて RESET パケットを作成し，ネット割込みが発生するように VMCS を変更する．これにより， CLOSE パケットを受信した直後に oops を起こす割込みを挿入することが可能になる．

提案手法により割込みを挿入し，報告されたバグレポートと同様の カーネル oops が発生した．

5.2.2 TCP に関するバグ

2.2.5 節で紹介した，TCP に関するバグを再現する．このバグを再現するために，バグが報告されている Linux 2.6.34 を domU としてを使用する．まず，**図 7** により，**図 8** のようにコネクション確立時の， ACK パケットのシークエンス番号が正しくないとカーネルパニックが生じることがわかる．次に，コードを調査すると，ip_local_out という関数で SYNACK パケットを送信し，ACK パケットを受信する前にシークエンス番号の異なる割込みを発生させることで，バグが再現することがわかる．

domU で TCP サーバを動かし，サーバが SYNACK パケットを送信するコードを実行した直後に vmcall を挿入し，操作を VMM に移す．この際，TCP の正しくない ACK パケットを持ったネット割込みを発生させるよう指定する．qemu-dm で TCP のパケットをエミュレートする関数に変更を加え，シークエンス番号が正しいシークエンス番号と異なる番号を持った ACK パケットを用意し，

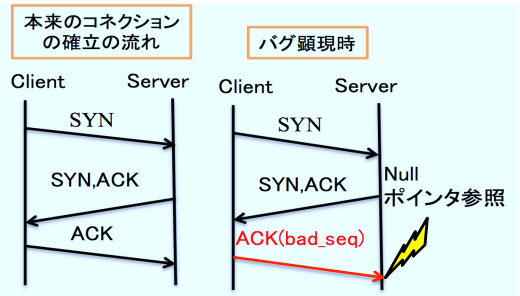

図 8 TCP におけるバグ

VMCS をネット割込みが発生するように変更する．この操作により SYNACK パケットを送信した直後にカーネルパニックを起こす割込みを挿入することが可能になる．

提案手法により割込みを挿入することで，カーネルパニックを引き起こした．

6. 関連研究

OS カーネルにおいてデバッグをする手法は数多く提案されている．しかし，これらの多くの手法ではタイミング依存のデバッグを十分に行うことは難しい．提案手法は，仮想化環境における割込みのエミュレート方法に着目したことで，指定したタイミングで指定した割込みを挿入することができるため，レポートされたバグの再現が可能になる

Linux のコードを静的に解析する研究は多くなされている．Palix らは，Linux 2.6 から Linux 2.6.33 までのソースコードのバグの調査を行っている [10]．バグを種類別に分けて，バグの数， バグの存在場所の変化，バグの存在期間を調べた．割り込み禁止の解除し忘れ，二重のロック獲得などの，典型的なバグを 13 種類定義し，それぞれの Linux のバージョンで調査を行い，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を示している．また，デバイスドライバのデバイス処理に関するバグを静的に解析し，自動で発見，修復するという研究も行われている [11]．この研究では，デバイスの故障により，期待するデータを持った割込みが発生しない場合，無限ループや配列への不適切なアクセスを修正している．しかし，これらの研究では，タイミング依存のバグを対象としておらず，期待していないデータを持った割込みへの対処などの，割込み処理に関するデバッグを行えない．

また，コードを網羅的に実行することにより OS カーネルのデバッグを行う研究も多く行われている．Chipounov らは，システムのコードを網羅的に実行する基盤ツール S2E[12] を提案している．この手法を用いることで，デバイスなどのシステム外部からの入力を具体的な値ではなくシンボルとして実行することで，コードを網羅的に実行できる．また，メモリアクセスを監視したり，競合状態を監視するプラグインを持っているため，広範囲のデバッグを

行うことができる．また，SymDrive[13] では，S2E を用いてデバイスドライバを網羅的に実行することができる．S2E の機能をデバイスドライバに特化し，またデバッグ用のフレームワークも追加している．この手法を用いることで，実際にデバイスを必要とせず検証することができる．これらは，コードを網羅的に実行することにより外部からの非決定的な入力を再現することが可能になる．さまざまな入力へのデバッグには対応できるが，タイミング依存のデバッグは対象としておらず，割込みを適切に禁止できているかなどの検査は行えない．

本研究と同様に非決定的なイベントに着目し，非決定的なイベントを再現する研究も行われている．Chow らは，OS の実行と，デバッグを行うマシンを分割することによって，OS の実行の非決定的なイベントに対するデバッグを可能にする手法 Aftersight [14] を提案している．この手法では，非決定的なイベントが起こるタイミング，種類をログに保存しておき，別のマシンで同様に実行することにより実行を再現する．また，King らは，ログを利用して，OS の実行を再現する手法 TTVM [15] を提案している．この手法では，OS の実行時に，定期的にチェックポイントを取り，CPU レジスタやメモリ，ディスク内容などをログに保存しておく．これにより，問題が発生した際に，任意のチェックポイントに戻り，同じ実行を再現することができる．これらの手法を用いることにより，バグが発現した際には，ログを元にバグを再現することができる．しかし，これらの手法では実際に割込みが発生しないとデバッグを行うことができない．

7. まとめ

本研究では，割込みのタイミングを意図的に制御可能にすることによりデバッグを支援する手法を提案した．提案手法では仮想マシンモニタを利用し，割込みのタイミングを意図的に制御する．仮想マシンモニタの，割込みをエミュレートするコードに変更を加えることで指定したコード位置に任意の割込みを発生させることができるようにする．

Xen 4.1.0 上に割込みを意図的に挿入する機構を作成した．domU の検査対象コードから呼び出されるハイパーコールをうけ，Xen で VMCS を書き換える仕組みを実装した．また，qemu-dm 上に，割込みのデータを準備する機構を作成した．ゲスト OS から割込み挿入依頼が発行されると，Xen で VMCS を書き換え，qemu-dm で割込みデータを用意することで，指定された割込みを挿入することができる．この機構を利用してタイマ割込み，キーボード割込み，ネットワーク割込みの挿入を行えるようにした．

この手法を用いることによって，バグレポートを参考に，バグの原因となる割込みを原因となる箇所に，意図的に挿入することにより，バグを再現することができる．Linux 公式サイトに報告されているバグのレポートを元にバグを含むカーネルを動かし，提案手法によりネットワーク割込み (TCP，DCCP) を挿入することで，再現できることを示した．

現状挿入できている割込みはタイマ割込み，キーボード割込み，ネットワーク割込みのみである．そのため，様々な割込み処理のデバッグを支援できるように，挿入できる割込みを増やさなければならない．

また，この手法はバグの再現だけでなく，バグの発見にも有用であると考えられる．現在の実装では，ハイパーコールをコードに直接埋め込んでいるため，利用者が入れるべき割込みの場所，種類を予想して利用しなければならない．しかし，ポリシーを作り，それに従って割込みの挿入を自動で行うようにすれば，自動でバグの検査を行うことができるようになる．

参考文献

[1] 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).
[2] Feng, W.: Making a case for efficient supercomputing, *Queue,1(7)*, pp. 54–64 (2003).
[3] Patterson, D.: Recovery Oriented Computing: A New Research Agenda for a New Century, *Proceedings of the 8th Intl. Symposium on High-Performance Computer Architecture* (2002).
[4] spinroot.com: SPIN, http://spinroot.com/spin/whatispin.html.
[5] 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)*, pp. 207–220 (2009).
[6] XenSource: Xen, http://www.xen.org/.
[7] Intel: IntelVirtualizationTechnology (Intel VT), http://www.intel.com/technology/virtualization/technology.htm.
[8] Hamano, J. C.: git.kernel.org, http://git.kernel.org/.
[9] Bellard, F.: QEMU, http://www.qemu.org/.
[10] 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)*, pp. 305–318 (2011).
[11] Kadav, A., Renzelmann, M. J. and Swift, M. M.: Tolerating Hardware Device Failures in Software, *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 1–16 (2009).
[12] Vitaly Chipounov, Volodymyr Kuznetsov, G. C.: S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems, *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems* (2011).
[13] Matthew J. Renzelmann, Asim Kadav, M. M. S.: SymDrive: Testing Drivers without Devices, *In Proceedings of the USENIX Symposium on Operating Systems De-*

sign and Implementatione, pp. 1–15 (2012).

[14] Peter M. Chen Jim Chow, T. G.: Decoupling dynamic program analysisfrom execution in virtual environments, *USENIX 2008 Annual Technical Conference*, pp. 1–14 (2008).

[15] King, S. T., Dunlop, G. W. and Chen, P. M.: Debugging operating systems with time-traveling virtual machines, *Proceedings of The USENIX Annual Technical Conference*, pp. 1–15 (2005).