情報処理学会論文誌 Vol. 51 No. 9 1816–1831 (Sep. 2010)

C プログラムの割込み競合の動的検出法

荒堀喜貴[†1] 権藤克彦[†1] 前島英雄[†2]

実用 C プログラムの割込み競合（割込み処理によるデータ競合）に対する新たな検出法を提案する．データ競合の検出法はこれまでに多数提案されている．しかし，それらのほとんどがスレッド競合（スレッド処理によるデータ競合）を対象としており，高精度なものも多数存在するが割込み競合を検出できない．一方，割込み競合を対象とする手法は数が少なく，検出精度も低い．そこで，我々は既存の高精度なスレッド競合検出法を改変し，割込み競合の高精度な検出を実現する．提案手法の要点は，割込み処理を擬似的なスレッド実行として扱い，割込み競合の検出問題をスレッド競合の検出問題に帰着させて解くことである．我々は提案手法に基づく検出ツールを実装し，評価用プログラムに適用して検出精度の評価実験を行った．その結果，提案手法は Sendmail や WU-FTPD を含む実用 C プログラムの割込み競合を高精度に検出できることが分かった．

Dynamic Interrupt-race Detection for C Programs

Yoshitaka Arahori,[†1] Katsuhiko Gondow[†1]
and Hideo Maejima[†2]

We present a novel approach to detect interrupt races (i.e. dataraces caused by interrupts) for real C programs. There have been a large amount of techniques proposed for datarace detection. However, most of them target thread races (i.e. dataraces caused by threads) and, despite of their high accuracy, cannot detect interrupt ones. Only a few techniques can deal with interrupt races but they are not precise. We propose to transform a precise thread-race detector so that it can detect interrupt races precisely. The main idea for the transformation is to treat every interrupt-handling as a pseudo-thread execution and reduce the problem of interrupt-race detection into that of thread-race detection. We have implemented an interrupt-race detector based on our approach, and measured its accuracy by applying it to several evaluation programs. The results show that our detector is precise enough to detect interrupt races caused by real C programs including Sendmail and WU-FTPD.

1. はじめに

1.1 背　景

データ競合は，主にマルチスレッド処理に混入するバグであり，(1) 同一メモリ位置に対して異なる 2 個のスレッドがアクセスを行い，(2) そのうちの少なくとも 1 つが WRITE アクセスであり，(3) それらのアクセス間に順序関係が強制されない，という 3 条件を満たすアクセスペアとして定義されている[1]．

データ競合は認識と再現が困難なバグである．なぜなら，データ競合を認識し再現するには，上記の条件を満たす 2 個のアクセスを特定のタイミングで発生させてデータの不整合（および，それにともなうプログラムの予期せぬ動作）を引き起こす必要があるからである．しかし，各スレッドによるメモリアクセスのタイミングは，入力データやスケジューリングに依存して複雑に変化するため，データの不整合を意図的に発生させることは難しい．このようにデータ競合は認識と再現が困難であるため，プログラムの開発効率を著しく低下させる．また，プログラムのテスト時にデータ競合の存在を認識できず，リリース後に不具合や脆弱性として報告されたケースもある[2)–4)]．したがって，データ競合の検出は重要な課題である．

このような事情から，様々なデータ競合検出手法が現在まで継続的に提案されている[5)–16)]．C プログラムに適用可能な手法も多数提案されているが，それらのほとんどがマルチスレッド処理のデータ競合（以降，スレッド競合と呼ぶ）の検出を目的としている．

一方，C プログラムでは，UNIX シグナルなどの非同期割込み処理もデータ競合の主要因となっている．これは，大部分の実用 C プログラムが非同期割込み処理を実装しているのに対し，非同期安全な処理の記述は制約が多く難しいからである[15)]．実際，Sendmail[17)]，WU-FTPD[18)]，BASH[19)] などの実用 C プログラムはすべてシグナル処理を実装しており，シグナル処理によるデータ競合が過去に（複数回）報告されている[15),20)–22)]．したがって，スレッド競合と同様，非同期割込み処理によるデータ競合（以降，割込み競合と呼ぶ）の検出も重要な課題である．

†1 東京工業大学大学院情報理工学研究科計算工学専攻
Department of Computer Science, Tokyo Institute of Technology
†2 東京工業大学大学院総合理工学研究科物理情報システム専攻
Department of Information Processing, Tokyo Institute of Technology

1.2 問題意識

スレッド競合の検出法に比べると非常に少数ではあるが，割込み競合の検出法もこれまでにいくつか提案されている[9),15)]．しかし，これらの手法は検出精度に重大な問題点がある．たとえば，Crocus[9)] はシグナルハンドラ内での非同期シグナル安全でないライブラリ関数の呼び出しを潜在的割込み競合として検出するが，共有変数へのアクセスに起因する競合を検出できない．DRACULA[15)] はグローバル変数へのアクセスに起因する割込み競合を検出するが，グローバル変数以外の共有オブジェクトへのアクセス競合を検出できない．また，マルチプロセスプログラムに対しては，ルートプロセス以外のプロセスを検査できない．

1.3 提案手法の要点

本研究では，実用 C プログラムの割込み競合を高精度に検出する手法を提案する．我々は Choi らのスレッド競合検出法[6)] を改変し，割込み競合の高精度な検出を可能にする．Choi らの手法は，対象プログラムの実行時に各オブジェクトに対する全アクセスをイベント履歴として記録する．スレッド競合の検出は，履歴中から競合条件を満たすアクセスの組を探索することで行う．この手法は既存のスレッド競合検出法の中でも比較的高精度であることが知られている[6),7)]．我々はこの手法を以下のように改変し，従来より高精度な割込み競合検出法を実現する：

- 割込み処理を擬似的なスレッド実行として扱い，割込み競合の検出問題をスレッド競合の検出問題に帰着させる．
- 検査コードをオブジェクト表を用いて実現し，対象プログラムと検査コードの互換性を維持する．

1.4 実験結果の要約

我々は提案手法に基づく割込み競合検出ツールを GCC[23)] の拡張として実装し，各種の実験を行った．実験の範囲内で，我々のツールは Sendmail[17)] や WU-FTPD[18)] を含む実用 C プログラムの割込み競合を高精度に検出できることが分かった．

1.5 本稿の構成

以下，2 章で，割込み競合を形式化し，動的検出アルゴリズムを示す．3 章で，我々の検出ツールの実装を示す．4 章で，検出ツールを評価用プログラムに適用して得られた各種の実験結果を示す．5 章で，関連研究を議論する．6 章で，結論と今後の展望を述べる．

2. 割込み競合の動的検出

本章ではまず，我々の競合検出法の適用対象を明確にする（2.1 節）．次に，割込み競合の判定条件を定義し（2.2 節），それに基づく検出アルゴリズムを示す（2.3 節）．最後に，我々の手法の妥当性（2.4 節）と制限事項（2.5 節）を述べる．

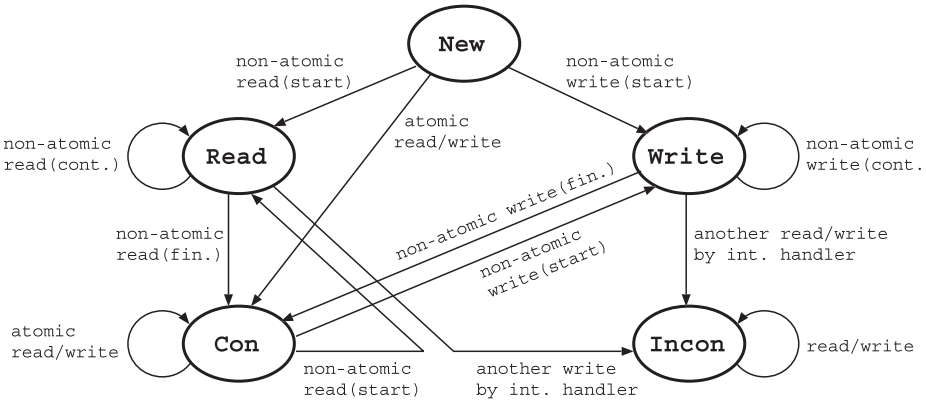

図 1　有効メモリオブジェクトの状態遷移
Fig. 1　State transitions of a valid memory object.

2.1 適用対象

提案する競合検出法の適用対象として，我々は次の条件を満たす C プログラムを想定する：

- **適用条件 1**：対象 C プログラムは割込み処理を行い，スレッド処理を行わない．割込み処理として想定するのは，非同期シグナルをシグナルハンドラで処理する場合である．
- **適用条件 2**：対象 C プログラムの実行時に，各有効メモリオブジェクトは，読み出しまたは書き込みアクセスによって，**図 1** に示す状態遷移を行う．

適用条件 1 は，我々の手法の目的が割込み競合の検出であることを意味する．既存の大部分の競合検出法とは異なり，我々はスレッド競合を検出しない．また，割込み処理とスレッド処理の両方を実行するプログラム（割込みハンドラを使用するマルチスレッドプログラム）も検出法の適用対象外とする（理由は 2.5.4 項を参照）．

適用条件 2 が示すとおり，我々の想定するプログラムでは，各メモリオブジェクトはアクセスを受けて図 1 の状態遷移を行うものとする．このうち，特定の遷移を引き起こすアクセスの組を割込み競合として扱う（2.2 節）．図 1 の各状態は次の意味を持つ：

- **New**：新規に割り当てられ，まだアクセスされていない状態．
- **Read**：アトミックでない読み出しアクセスを受けている途中の状態．オブジェクトの

値は整合性を有するが，読み出し途中の値は整合性を欠く．

- **Write**：アトミックでない書き込みアクセスを受けている途中の状態．オブジェクトの値は整合性を欠く．
- **Con**：アクセスが完了し，オブジェクトの値が整合性を有する状態（consistent）．
- **Incon**：あるアトミックでないアクセスの途中で他の割込み処理（のアクセス）に割り込まれることにより，オブジェクトの値または読み出し途中の値が整合性を失ったことのある状態（inconsistent）．

アクセスがアトミックであるとは，アクセスが割込み処理中のアクセスに割り込まれないことを意味する．アクセスのアトミック性は主に，アトミックなメモリアクセス命令や割込みマスクによるハンドラの実行制御で実現される．

状態 New のオブジェクトは，アトミックな読み出しまたは書き込みアクセスを受けると状態 Con に遷移する．アトミックでない読み出しまたは書き込みアクセスが開始（start）されると状態 Read または状態 Write に遷移する．

状態 Read のオブジェクトは，読み出しの途中で，別の（割込みハンドラによる）書き込みアクセスに割り込まれる可能性がある．割り込まれた場合，状態 Incon に遷移する．割り込まれなかった場合，読み出しの継続中（cont.）は状態 Read に滞在し，読み出し完了（fin.）とともに状態 Con に遷移する．なお，状態 Read のオブジェクトは他の読み出しアクセスに割り込まれても状態 Incon には遷移しない（状態 Read に滞在する）．

状態 Write のオブジェクトは，書き込みの途中で，別の（割込みハンドラによる）アクセスに割り込まれる可能性がある．割り込まれた場合，状態 Incon に遷移する．割り込まれなかった場合，書き込みの継続中（cont.）は状態 Write に滞在し，書き込み完了（fin.）とともに状態 Con に遷移する．

状態 Con のオブジェクトは，アトミックな読み出しまたは書き込みアクセスを受けると状態 Con に遷移する．アトミックでない読み出しまたは書き込みアクセスが開始（start）されると状態 Read または状態 Write に遷移する．

状態 Incon のオブジェクトは以後どのようなアクセスを受けても同状態に滞在し続ける．

2.2　割込み競合の判定条件

前節の適用条件を満たすプログラムに対し，我々はオブジェクトの状態を Read または Write に遷移させるアクセスとそこから状態 Incon に遷移させるアクセスの組を直接的な割込み競合として扱う．また，プログラムのある実行で状態 Con への遷移を引き起こすアクセスの組であっても，別の実行で状態 Incon への遷移を引き起こしうる[⋆1]ものは潜在的な割込み競合とする．

本節では，これらの直接的または潜在的な割込み競合を一定の精度で検出するための手段として，割込み競合の判定条件を定義する．まず，準備として，アクセスイベントを以下のように定義する．

定義 1（アクセスイベント） アクセスイベント e は 5 つ組 (m, i, M, a, s) である．ここで，イベントの構成要素は以下のとおり：

- m はアクセス対象のメモリオブジェクト
- i はアクセスを行う割込みスレッドの ID
- M はアクセス時に `mask` 済みの割込みの集合
- a はアクセスタイプ（WRITE または READ）
- s はアクセス実行位置（ファイル名と行番号）

この定義は，スレッド競合の検出を目的とする Choi らのイベント定義[6] を割込み競合の検出に向けて改変したものである．

割込み競合の検出問題をスレッド競合の検出問題に帰着させるために，我々は割込みハンドラの実行を擬似的なスレッドの実行として扱う．この擬似スレッドを割込みスレッドと呼ぶ．発生したアクセス e の割込みスレッド ID（$e.i$）は以下の方針で決定する：

- 割込み処理中である場合，$e.i$ は処理中の割込み番号である．
- 割込み処理中でない場合，$e.i$ はどの割込み番号とも一致しない特別な番号 i_N である．

この方針から分かるように，我々は割込み処理中でない場合も，割込み番号 i_N のハンドラ（に対応する擬似スレッド）を実行していると考える．なお，割込みスレッド ID に単純な割込み番号を使用することの得失については 2.5 節で議論する．

次に，割込み競合の判定条件を以下のように定義する．

定義 2（割込み競合の判定条件） 2 個のアクセスイベント e_1，e_2（この順序で発生が観測されたとする）に対し，

⋆1 割込みマスクによるハンドラ実行制御に誤りがある場合，割込みの発生タイミングに依存して，同じアクセスの組であっても異なる状態遷移を引き起こす可能性がある．

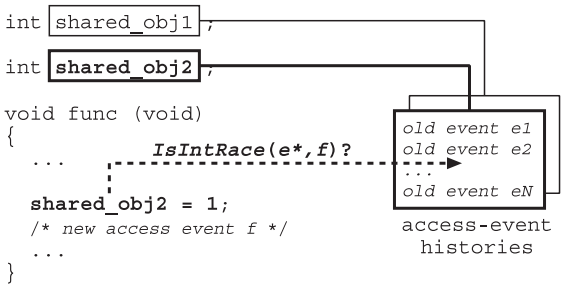

図 2　アクセスイベント履歴に基づく動的データ競合検出
Fig. 2　Dynamic datarace detection based on access event histories.

$$
\begin{aligned}
&\mathit{IsIntRace}(e_1, e_2)\\
&\Leftrightarrow (e_1.m = e_2.m) \wedge (e_1.i \neq e_2.i)\\
&\quad \wedge (e_1.a = \mathrm{WRITE} \vee e_2.a = \mathrm{WRITE})\\
&\quad \wedge (e_2.i \notin e_1.M) \wedge (e_2.i \neq i_N)
\end{aligned}
$$

すなわち，(1) 同一のメモリオブジェクトに対して異なる割込みスレッドがアクセスを行い，(2) そのうちの少なくとも 1 つが WRITE アクセスであり，(3) 割込みマスクの不備により，先に観測されたアクセスが後に観測されたアクセスに割り込まれる（可能性がある）場合，我々はそれらのアクセスの組を割込み競合と判定する．ただし，割込みスレッド ID が i_N であるアクセスは割込みを処理していない間に発生したアクセスであるため，他のアクセスを割り込むことがない（条件 $e_2.i \neq i_N$）．

2.3　動的競合検出アルゴリズム

本節では，前節で定義した競合判定条件に基づき動的に割込み競合を検出する手法を示す．我々は Choi らのスレッド競合検出手法[6] を改変して割込み競合を検出できるようにする．Choi らの手法と同様，プログラム実行時に各メモリオブジェクトに対して競合検出用のイベント履歴を関連付け，発生したアクセスイベントを順次記録していく（**図 2**）．各記録の直前ではイベント履歴を探索し，現在のアクセスイベントと競合する過去のイベントを探す．ここで，Choi らの手法と異なり，我々は競合判定条件として *IsIntRace* を使用する．すなわち，現在のアクセスイベントと条件 *IsIntRace* を満たす過去のイベントを履歴内で探す．見付かれば，それらのイベントの組を割込み競合として報告し，見付からなければ，現在のイベントを履歴に記録する．

Choi らは実行進行にともなうイベント履歴のサイズ増加および検査オーバヘッドを抑制するために，スレッド競合の検出に必要なアクセスイベントだけを選別し記録する最適化手法を開発した．彼らは 2 個のアクセスイベント e_1，e_2 の間にスレッド競合に対する脆弱性の序列を表現する半順序関係 $e_1 \sqsubseteq e_2$（weaker-than 関係）を定義し，より脆弱なアクセスイベントのみを履歴に残せばよいことを示した（weaker-than 定理）．

我々は割込み競合の検出における weaker-than 関係/定理を新たに定義/証明し，割込み競合の検出においても履歴サイズと検査オーバヘッドを低減できるようにする．まず，アクセスイベント e_1，e_2 に対し，weaker-than 関係 $\sqsubseteq_I$ を次のように定義し，割込み競合に対する脆弱性の序列を表現する．

定義 3（割込み競合に関する weaker-than 関係）　アクセスイベント e_1，e_2 に対し，

$$
\begin{aligned}
&e_1 \sqsubseteq_I e_2\\
&\Leftrightarrow (e_1.m = e_2.m) \wedge (e_1.a \sqsubseteq_a e_2.a)\\
&\quad \wedge (e_1.i \sqsubseteq_i e_2.i) \wedge (e_1.M \subseteq e_2.M)
\end{aligned}
$$

ここで，アクセスタイプと割込みスレッドの weaker-than 関係を以下のように定義する．

定義 4　アクセスタイプ a_1，a_2 に対し，

$$a_1 \sqsubseteq_a a_2 \Leftrightarrow (a_1 = a_2) \vee (a_1 = \mathrm{WRITE})$$

定義 5　割込みスレッド ID i_1，i_2 に対し，

$$i_1 \sqsubseteq_i i_2 \Leftrightarrow (i_1 = i_2) \vee (i_1 = i_\perp)$$

定義内の $i_\perp$ はイベント履歴内でのみ出現する特別な割込みスレッド ID であり，異なる複数の割込みスレッドによる類似のアクセスイベントを 1 個にまとめる役割を果たす．たとえば，イベント履歴に過去のイベント p のエントリが存在し，新たなイベント f が発生して $p.m = f.m$，$p.a = f.a$，$p.i \neq f.i$，$p.M = f.M$ が成り立つ場合を考える．この場合，我々は p と f を割込みスレッド ID だけが異なる類似のイベントであると考え，f のエントリを履歴に追加するかわりに p のエントリのフィールド $p.i$ を $i_\perp$ に変更する．このようにして $i_\perp$ は複数の割込みスレッドによる類似のアクセスイベントを表現する．したがって，$i_\perp$ はどの割込みスレッド ID とも一致しない．

$p_1 \sqsubseteq_I p_2$ が成り立つ場合，アクセスイベント p_1 は p_2 よりも割込み競合に対して脆弱である（将来 p_1 と競合しうるイベントの集合は，p_2 と競合しうるイベントの集合を包含す

る）．次の weaker-than 定理はこのことを表現する．

定理 1（割込み競合に関する weaker-than 定理） 過去のイベント p_1，p_2 と未来のイベント f に対し，

$$p_1 \sqsubseteq_I p_2 \Rightarrow$$
$$(\mathit{IsIntRace}(p_2, f) \Rightarrow \mathit{IsIntRace}(p_1, f)).$$

証明　$p_1 \sqsubseteq_I p_2$ と $\mathit{IsIntRace}(p_2, f)$ が成り立つと仮定する．$\sqsubseteq_I$ の定義より，(1) $p_1.m = p_2.m$，(2) $p_1.a \sqsubseteq_a p_2.a$，(3) $p_1.i \sqsubseteq_i p_2.i$，(4) $p_1.M \subseteq p_2.M$ が成り立つ．また，$\mathit{IsIntRace}$ の定義より，(5) $p_2.m = f.m$，(6) $p_2.a = \mathrm{WRITE}$ または $f.a = \mathrm{WRITE}$，(7) $p_2.i \neq f.i$，(8) $f.i \notin p_2.M$，(9) $f.i \neq i_N$，が成り立つ．(1) と (5) により，(10) $p_1.m = f.m$ を得る．$p_2.a = \mathrm{WRITE}$ の場合，(2) より $p_1.a = \mathrm{WRITE}$ である．したがって，(6) より，(11) $p_1.a = \mathrm{WRITE}$ または $f.a = \mathrm{WRITE}$ が成り立つ．$p_1.i \neq i_\perp$ の場合，(3) より $p_1.i = p_2.i$ が成り立ち，したがって，(7) より，$p_1.i \neq f.i$ が成り立つ．$p_1.i = i_\perp$ の場合，ただちに $p_1.i \neq f.i$ が成り立つ．よって，いずれにしても，(12) $p_1.i \neq f.i$ が成り立つ．(4) と (8) により，(13) $f.i \neq p_1.M$ を得る．以上，(9) から (13) より，$\mathit{IsIntRace}(p_1, f)$ が成り立つ．（証明終了）

割込み競合に関する weaker-than 関係/定理を活用すると，競合検出に必要なアクセスイベントだけを選別してイベント履歴に記録する最適化が可能となる．すなわち，イベント履歴のサイズ低減と履歴探索の高速化を実現できる．たとえば，あるイベント e_2 が発生したときに $e_1 \sqsubseteq_I e_2$ を満たす過去のイベント e_1 が履歴中に存在する場合，e_2 の記録は省略することができる．なぜなら，weaker-than 定理により，将来 e_1 と競合しうるイベントの集合が e_2 と競合しうるイベントの集合を包含しているため，e_1 さえ履歴に残しておけば e_2 と競合するイベントも検出できるからである．

次に，割込み競合検出アルゴリズムの詳細を説明する．Choi らの手法と同様，我々はイベント履歴を trie 木で表現する．ただし，trie 木のエッジとノードは割込み競合の検出に必要な情報を表現する．**図 3** は，あるメモリオブジェクト m_1 に関連付けられたイベント履歴の例である．各エッジは `mask` 済みの割込み番号を保持し，各ノードは割込みスレッド ID とアクセスタイプを保持する．各ノードとそこに至るパスが 1 個のアクセスイベントを表現する．たとえば，ノード e_1 とそこに至るパスは，$e_1.m = m_1$，$e_1.i = i_N$，$e_1.a = \mathrm{WRITE}$，$e_1.M = \{i_1, i_2\}$ を満たすアクセスイベント e_1 を表現している．図 3 のイベント履歴には，他に 2 個のイベント e_2，e_3 が記録されている．割込みスレッド ID とアクセスタイプを保持しない（空白の）ノードは，アクセスイベントを表現しない中間ノードである[⋆1]．

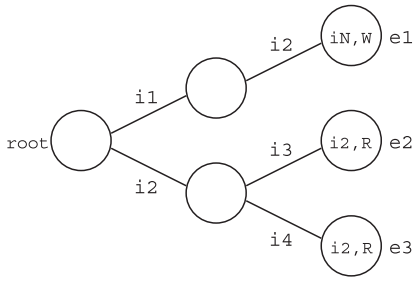

図 3　trie 木を用いた割込み競合検出用イベント履歴の表現
Fig. 3　Trie-based representation of an event history for interrupt-race detection.

割込み競合の検出はイベント履歴の trie 木を探索することで行う．あるメモリオブジェクトに対してアクセスイベント f が発生したとき，そのオブジェクトのイベント履歴に対して以下のステップを実行する：

- **Step 1**：f **より脆弱な過去のイベントの検索**
  （1）割込みマスク $f.M$ 内の割込み番号を探索キーにして深さ優先探索を行い，出会ったノード p について条件 $p.a \sqsubseteq_a f.a \wedge p.i \sqsubseteq_i f.i$ の成立を判定する．
  （2）成立するノード p が見付かった場合，$p \sqsubseteq_I f$ が成り立つため，f を履歴に記録せずに検出アルゴリズムを終了する．
- **Step 2**：f **と競合する過去のイベント** p **の検索**
  trie 木の全ノードを深さ優先探索で訪問し，出会ったノード p（と p へのエッジ i_p）について，
  Case 1：$i_p = f.i$ の場合，$f.i \in p.M$ が成り立つ，すなわち，p およびその子ノード群は f による割込みから保護されている．したがって，p をルートとする部分木の探索をスキップして，深さ優先探索を継続する．
  Case 2：$i_p \neq f.i$ の場合，条件 $(p.a = \mathrm{WRITE} \vee f.a = \mathrm{WRITE}) \wedge (p.i \neq f.i)$ の成立を判定する．成立する場合，$\mathit{IsIntRace}(p, f)$ が成り立つので，割込み競合 (p, f) を報告して検出を終了する．成立しない場合，p の子ノードを深さ優先探索する．

⋆1 ただし，中間ノードがイベントを表現する場合もある．たとえば，図 3 の履歴に $e_4.i = i_1$，$e_4.a = \mathrm{READ}$，$e_4.M = \{i_2\}$ なるイベント e_4 を記録すると，中間ノード（e_2 と e_3 の親）がイベント e_4 を表現する．

- **Step 3：イベント f の記録**
  $f.M$ 内の割込み番号をキーにして深さ優先探索を行う．
  Case 1：$p.M = f.M$ を満たすノード p が存在しない場合，新たに f のノード（とそこに至るパス）を追加する．
  Case 2：$p.M = f.M$ を満たすノード p がすでに存在する場合，p がイベントを表現しない中間ノードなら，$p.i$，$p.a$ にそれぞれ $f.i$，$f.a$ の値を設定する．p が過去のイベントを表現するノードなら，p と f は類似する（$p.M = f.M$）イベントなので，両者の記録を 1 つにまとめる．すなわち，$p.i \neq f.i$ なら $p.i$ に $i_\perp$ を設定し，$p.a \neq f.a$ なら $p.a$ に WRITE を設定する．
- **Step 4：f より強い過去のイベント p の削除**
  深さ優先探索で全ノードを訪問し，新たに記録したイベント f より割込み競合に強いイベント（$f \sqsubseteq_I p$ かつ $f \neq p$ を満たす p）を削除する．

割込み競合の weaker-than 定理に基づく最適化は検出アルゴリズムの **Step 1** と **Step 4** で実行する．**Step 1** では，weaker-than 定理に基づき，競合検査に不要なアクセスイベントを無視する（**Step 2** 以降で処理するのは **Step 1** で必要なイベントとして選別したものだけ）．実用プログラムの検査においては，大部分のイベントが無視され，検出アルゴリズムは **Step 1** で早期終了する．**Step 4** では，weaker-than 定理に基づき，競合検査に不要なアクセスイベントを履歴から削除する．このようにして，検出アルゴリズムは履歴サイズと検査時間の削減を実現している．

2.4　検出法の妥当性

割込み競合の判定条件（および，それに基づく検出アルゴリズム）は，従来手法[9),15)]より高精度な検出を可能にする．本節では，このことを検出法の妥当性として，実際の検出例を交え定性的に説明する．

割込み競合の判定条件 *IsIntRace* を使用すると，**図 4** のプログラム（base-3.0）が引き起こすような典型的な割込み競合を検出できる．bash-3.0 はマルチプロセスのシェルプログラムであり，ユーザが入力したコマンドを実行する．実行したコマンドは bashhist.c で定義された関数 `really_add_history` を用いてコマンド履歴に追加する．また，履歴中のコマンド行数を変数 `history_lines_this_session` で管理する．bash-3.0 はシグナル SIGHUP を受信すると，対応するハンドラを起動し，bashhist.c で定義された関数 `maybe_append_history` を実行する．ここで，ルートプロセスが bashhist.c の 702 行目で変数 `history_lines_this_session` に書き込みを行っている間に SIGHUP が発生し，対応するハンドラが関数 `maybe_append_history` を呼び出して bashhist.c の 314 行目で同一の変数の値を読み出す場合を考える．この場合の両アクセスは直接的割込み競合である．なぜなら，変数 `history_lines_this_session` は先の書き込みアクセスにより（状態 Con から）状態 Write に遷移し，後の読み出しアクセスにより状態 Incon に遷移するからである．*IsIntRace* はこの直接的割込み競合を検出できる．なぜなら，先の書き込みを e_1 とし後の読み出しを e_2 とすると，$e_1.m = e_2.m =$ `history_lines_this_session`，$e_1.i = i_N$，$e_2.i =$ SIGHUP，$e_1.a =$ WRITE，SIGHUP $\notin e_1.M$ であり，$\mathit{IsIntRace}(e_1, e_2)$ が成立するからである．次に，プログラムの別の実行で書き込みアクセス e_1 の完了後に読み出しアクセス e_2 が実行された場合を考える．この場合の両アクセスの組は潜在的割込み競合であるが，*IsIntRace* はこの競合を検出できる．なぜなら，この場合も同様に $\mathit{IsIntRace}(e_1, e_2)$ が成立するからである．

一方，書き込みアクセス e_1 が適切なシグナルマスクで保護されている場合（アクセスイベント e_1 の発生時に SIGHUP がマスクされている場合），上述の割込み競合は発生しない．この

bashhist.c

```
int
maybe_append_history (filename)
     char *fileline;
{
  ...
  result = EXECUTION_SUCCESS;
  if (history_lines_this_session
       && (...))
    {
      ...
    }
  ...
}
```

```
static void
really_add_history (line)
     char *line;
{
  hist_last_line_added = 1;
  add_history (line);
  history_lines_this_session++;
}
```

図 4　割込み競合を起こすプログラム例（bash-3.0）
Fig. 4　Example program (bash-3.0) with an interrupt race.

場合，*IsIntRace* は競合を誤検出しない．なぜなら，SIGHUP $\in e_1.M$ かつ $e_2.i =$ SIGHUP であり，したがって，$IsIntRace(e_1, e_2)$ の条件 $e_2.i \notin e_1.M$ が成立しないからである．

このように，我々の割込み競合の判定条件 *IsIntRace* は図 4 に示す典型的な（直接的または潜在的）割込み競合の検出に有効である．

これに対し，従来手法の 1 つである Crocus[9] は上述の競合を検出できない．なぜなら，Crocus の検出方式は，シグナルハンドラ内での非同期シグナル安全でない関数の呼び出しを潜在的競合として検出するというものであり，共有変数へのアクセスに起因する競合を検出できないからである．

一方，DRACULA[15] は，グローバル変数上の割込み競合を検出する方式であるため，上述の競合を検出できる．ただし，DRACULA が検査対象とするのはルートプロセスのみであるため，子プロセス内で発生する競合は検出できない．また，DRACULA はグローバル変数以外の共有オブジェクトへのアクセス競合を検出できない．我々の手法にこれらの制限はない（全プロセス/全オブジェクトが検査対象）．

以上より，我々の提案手法は従来手法より高精度な検査が可能であると考えられる．なお，4 章では，評価用プログラムを用いて検出精度をより詳細に計測する．

2.5 制 限 事 項

前節では提案手法が従来手法より高精度であることを述べたが，直接的または潜在的割込み競合（2.2 節）の健全かつ完全な検出が可能であるわけではない．そこで本節では，我々の手法の制限事項（特に，検出精度に関する問題点）を述べる．

2.5.1 割込み競合の判定条件の問題点

我々の割込み競合判定条件は直接的または間接的割込み競合の検出に失敗する場合がある．なぜなら，我々の判定条件には以下の問題点があるからである．

- **問題点 1（順序の仮定）**：アクセスイベントが特定の観測順序と同じ順序で発生すると仮定している．
- **問題点 2（コンテキストの抽象化）**：割込み処理中のハンドラのコンテキストの抽象化が粗い（割込みスレッド ID の表現として単純な割込み番号を使用している）．

問題点 1 が割込み競合の検出漏れを引き起こす例として，図 4 のプログラムで bashhist.c の 702 行目の変数 `history_lines_this_session` の書き込みが観測される前に SIGHUP が発生し，対応するハンドラの実行で 302 行目の同変数の読み出しが先に観測された場合を考える．この場合，先に観測された読み出しを e_1 とし後に観測された書き込みを e_2 とすると，$e_2.i = i_N$ であるため，条件 $IsIntRace(e_1, e_2)$ が成立しない．したがって，*IsIntRace* は両アクセスが観測とは逆の順序で発生した場合の潜在的割込み競合を検出できない．

しかし，実用プログラムの検査において，このような検出漏れが発生する場合は稀であると考えられる．実際，実用プログラムでは，割込みハンドラよりも先に非ハンドラが共有変数をアクセス（初期化）する場合がほとんどである．また，多くの場合において，非同期割込みが発生するのは，非ハンドラが様々な実行パスで共有変数を複数回アクセスした後である．したがって，我々は問題点 1 の検出漏れが発生する場合は少ないと考え，現状は対策を行っていない．

ただし，検出漏れの低減が技術的に不可能であるわけではない．たとえば，$e_1.i \neq i_N$ かつ $e_2.i = i_N$ である場合に限り，観測とは逆の順序でのアクセスの発生を想定して条件 $IsIntRace(e_2, e_1)$ の追加判定を行えば，検出漏れを低減できる．

問題点 2 により，主に次の 2 つの場合で割込み競合の検出漏れが発生しうる．

- **場合 1（割込み処理の再入）**：ある割込みの処理中に同一の割込みが発生して同一のハンドラが起動される場合．
- **場合 2（割込み処理のネスト）**：ある割込みの処理中に別の割込みが発生して対応するハンドラが起動され，かつ，前者の割込みを受けるアクセスが後者の割込みをマスクしている場合．

場合 1 の例として，SIGINT ハンドラがある変数に書き込み e_1 を行っている途中で SIGINT が発生し，同一のハンドラが同じ変数に書き込み e_2 を行う場合を考える．この例の両アクセスの組は直接的割込み競合であるが，$e_1.i = e_2.i =$ SIGINT であるため，条件 $IsIntRace(e_1, e_2)$ が成立しない．

場合 2 の例として，以下のシナリオを考える：

- ある共有変数へのアクセス e_1 が発生する．このときに，SIGHUP がマスクされており SIGINT がマスクされていない（SIGHUP $\in e_1.M$ かつ SIGINT $\notin e_1.M$）．
- アクセス e_1 の途中で SIGINT が発生し，SIGINT ハンドラが同一の変数をアクセスせず SIGHUP のマスクを解除する．
- マスク解除直後に SIGHUP が発生し，SIGHUP ハンドラが同一の変数に書き込み e_2 を行う（$e_2.i =$ SIGHUP）．

この例の両アクセスの組は直接的割込み競合であるが，$e_2.i \in e_1.M$ であるため，条件 $IsIntRace(e_1, e_2)$ が成立しない．

上記の 2 つの場合における検出漏れへの対策は今後の重要な課題である．なぜなら，一部の実用プログラムでは，高度な即応性が要求されるなどの理由で，割込み処理の再入やネス

トが必要となるからである．

検出漏れを低減させる方法の 1 つは，割込み処理のコンテキストをより詳細に表現したものを割込みスレッド ID として使用することである．たとえば，割込み番号のスタックを割込みスレッド ID として使用し，割込み処理のネスト（どのハンドラがどのハンドラを割り込んでいるか）を追跡管理する方法などが考えられる．ただし，コンテキストの詳細化による検査精度の向上と検査時間の増大はトレードオフの関係にある[⋆1]ため，適切な妥協点を選ぶことが重要である．

2.5.2　ハンドラ登録前のアクセスに起因する誤検出

対象プログラムが割込みハンドラの登録前に共有変数を初期化する場合，我々の検出法は，初期化時のアクセスと割込み処理時の（ハンドラによる）アクセスが割込み競合を引き起こすと誤検出してしまう．実用プログラムではハンドラ登録前に共有変数を初期化することが頻繁に行われるため，この誤検出への対処は重要である．

そこで，我々の検出器は現状，ハンドラ登録関数（`signal` や `sigaction` など）が最初に呼び出されるまで競合検査を行わない（イベント履歴を更新しない）という制御を行い，誤検出を低減させている．ただし，この対処法は完全ではなく，より高度な制御（たとえば，happens-before 解析の導入など）に基づく誤検出の低減は今後の重要な課題である．

2.5.3　イベントエントリの統合による悪影響

競合検出アルゴリズムは割込みスレッド ID $i_{\perp}$ を用いてイベント履歴内の類似のエントリを 1 つに統合するが，この最適化は競合検出後のデバッグ効率を低下させる恐れがある．なぜなら，類似のイベントを統合することによって，競合を構成する一方のイベントの割込みスレッド ID を特定できなくなるからである．

ただし，このデバッグ効率の低下はそれほど深刻ではないと考えられる．なぜなら，類似エントリが統合された場合でも，競合検出時に一方のイベントの割込みスレッド ID は必ず特定できるからである．特に，典型的な割込み競合は非ハンドラのアクセスを割込みハンドラのアクセスが割り込む場合に発生する．この場合，デバッグに有用なのはハンドラの割込み番号であり，この値は正確に特定できる[⋆2]ため，デバッグ効率は低下しない．一方，割込みハンドラを別のハンドラが割り込む場合の競合については，割り込まれる側の割込みスレッド ID が $i_{\perp}$ に縮退している場合に元の ID を特定できない．しかし，その場合もデバッグ効率の低下は深刻でない．なぜなら，割り込まれる側のマスクの設定状況やアクセス対象の共有オブジェクトなどの情報から，割り込まれる側のハンドラを容易に特定できるからである．以上の理由により，我々は現状，イベントの統合によるデバッグ効率の低下について対策を行っていない．

2.5.4　マルチスレッドプログラムへの適用

我々の競合検出法は，2 つの理由により，マルチスレッドプログラムを適用対象外とする．第 1 の理由は，検出法の目的が割込み競合の検出であり，スレッド競合の検出ではないからである．第 2 の理由は，マルチスレッドプログラムでは，割込みハンドラを用いて非同期的に割込みを処理することが比較的少ないと予想されるからである．POSIX に準拠するスレッドライブラリでは，割込みハンドラよりも安全かつ容易な処理手段として `sigwait`（および，その派生関数）が提供されているため，それらの使用が一般的であると考えられる．

3.　検出ツールの実装

本章では，検出ツールの実装として，検査コードの挿入法と処理内容を説明する．我々のツールは対象プログラムのコンパイル時に検査コードを挿入し，生成されたコードを実行することで競合を検出する．ここで，対象コードと検査コードの互換性を維持するために，我々はオブジェクト表を用いて検査コードを実現する（拡張ポインタ表現は使用しない）．ただし，オブジェクト表に基づく検査コードは非同期割込み処理中に競合状態を引き起こしてしまう．この問題を回避するために，我々は遅延検査と呼ぶ検査手法を導入する．

3.1　オブジェクト表に基づく検査コード

我々のツールが挿入する検査コードには，オブジェクト追跡コードとアクセス検査コードの 2 種類がある．

3.1.1　オブジェクト追跡コード

オブジェクト追跡コードは，各メモリオブジェクトとそのイベント履歴の関連付けを heap 領域上の表を用いて実現する．この表をオブジェクト表と呼ぶ．オブジェクト追跡コードは，各オブジェクトの割当て/解放の実行位置に挿入する．たとえば，`malloc`/`free` の呼び出しに対し，次の挿入を行う：

```
  p = malloc (size);
# if (p != NULL) reg_obj (p, size);
```

⋆1 実際，割込みスレッド ID の表現として単純な割込み番号を使用した場合，検出漏れが発生する反面，検査時に ID を参照する各種の計算（$e_1.i \neq e_2.i$ や $e_2.i \notin e_1.M$ など）は単純な整数のビット演算として高速に実行できる．これに対し，割込みスレッド ID を割込み番号のスタックで表現した場合，検出漏れを低減できる反面，検査時の ID 参照の計算コストはビット演算に比べ大幅に増加してしまう．

⋆2 $i_{\perp}$ ではない．なぜなら，競合検出の時点ではハンドラのアクセスはまだ履歴に登録されていないからである（2.3 節）．

```
    ...
    free (p);
# if (p != NULL) unreg_obj (p);
```

ここで，#で始まる行がオブジェクト追跡コードである．関数 reg_obj は新しく割り当てられたオブジェクトのベースアドレス，サイズ，イベント履歴からなるエントリ[⋆1]をオブジェクト表に登録する．一方，関数 unreg_obj は解放されたオブジェクトのエントリをオブジェクト表から削除する．

static 領域または stack 領域上のオブジェクトについても，オブジェクト追跡コードを挿入する：

```
    char g_buf[64];
# void init_global_objs (void) {
#   reg_obj (&g_buf[0], sizeof(g_buf));
# }

    void func (void) {
      char buf[32];
#     reg_obj (&buf[0], sizeof(buf));
      ...
#     unreg_obj (&buf[0]);
      return;
    }
```

ここで，関数 init_global_objs はプログラムの開始直後に 1 度だけ呼ばれて各グローバル変数とそのイベント履歴の関連付けをオブジェクト表に登録する．

なお，sig_atomic_t 型のオブジェクトは，アトミックにアクセスされることが保証される[⋆2]ので，誤検出防止のためにオブジェクト追跡コードを挿入しない．

我々のプロトタイプ実装では，オブジェクト表を heap 領域上の splay 木[24] として実現し，各エントリをオブジェクトのベースアドレスとサイズの組で索引付けしている．splay 木を採用した理由は，(1) 実装が単純であり，かつ，(2) エントリアクセスの時間的局所性を利用してイベント履歴の取得に要する償却計算量を低く抑えられるからである．

3.1.2　アクセス検査コード

アクセス検査コードは，アクセスイベント発生時に，(1) オブジェクト表を検索してアクセス対象オブジェクトのイベント履歴を取得し，(2) イベント履歴を検索して現在のイベントと競合する過去のイベントを探す．アクセス検査コードは，各メモリアクセスの実行位置に挿入する．たとえば，変数アクセスに対し，次の挿入を行う：

```
# chk_acc (&var, sizeof(var), WRITE);
  var = 0;
```

ここで，関数 chk_acc はまず，アクセス対象オブジェクト var のベースアドレス&var とサイズ sizeof(var) の組を探索キーとしてオブジェクト表を走査し，オブジェクトに関連付けられたイベント履歴を取得する．次に，イベント履歴を走査して現在のイベント（変数 var の WRITE）と競合する過去のイベントを探す．このときに，2.3 節のアルゴリズムを使用する．

変数アクセス以外にも，ポインタのデリファレンスや配列/構造体の要素アクセスなどすべてのメモリアクセスに対して，アクセス検査コードを挿入する．

メモリ読み書き用の外部ライブラリ関数の呼び出しは競合検査付きのラッパ関数に置換する．たとえば，関数 read の呼び出し：

```
read (fd, buf, count);
```

はラッパ関数 read_wrapper に置換する：

```
# read_wrapper (fd, buf, count);
```

ここで，ラッパ関数 read_wrapper は元の関数 read を呼び出す前に競合検査を行う：

```
ssize_t
read_wrapper (int fd, void *buf,
              size_t sz)
{
  chk_acc (buf, sz, WRITE);
  return read (fd, buf, sz);
}
```

外部ライブラリ関数が非同期割込み安全でない場合は，メモリ読み書き用の関数でなくてもラッパに置換する．ラッパ関数は元の関数が管理する内部データへのアクセスを擬似的に生成し検査する．その結果，元の関数が引き起こす競合を検出することができる．たとえば，ライブラリ関数 malloc はメモリ読み書き用の関数ではないが非同期割込み安全でないため，ラッパ関数 malloc_wrapper に置換する：

```
void *
```

⋆1 我々のプロトタイプ実装では，各エントリは追加情報として，オブジェクトの領域種別（static，heap，stack など）や割当て位置（ファイル名と行番号）などを含む．これら追加情報は，競合検出後のデバッグに役立つ．

⋆2 ただし，ポインタのエイリアスやキャストによってこの保証が崩れる場合もある．そのため，我々のツールは sig_atomic_t 型のオブジェクトに対し，例外的にオブジェクト追跡コードを挿入するオプションを提供している．

```
malloc_wrapper (size_t size)
{
  chk_acc (MALLOC_INTERN, 1, WRITE);
  return malloc (size);
}
```

ここで，関数 chk_acc の呼び出しは，malloc の管理する内部データ（MALLOC_INTERN）に対するアクセスを擬似的に生成して検査している．この検査により，割込みハンドラ内で malloc を呼び出した場合に発生する競合が検出可能となる．

3.1.3 検査コードと対象コードの互換性

オブジェクトとイベント履歴の関連付けをオブジェクト表で実現することにより，検査コードは対象コードとの互換性を高度に維持できる．なぜなら，オブジェクト表に基づく関連付けはポインタの内部表現を変更しないからである．

一方，オブジェクトと履歴を関連付ける代表的手法として，拡張ポインタ表現[⋆1]を使用する手法がある．拡張ポインタ表現はポインタの内部表現を複数ワードに拡張したものであり，拡張ワードがポイント先のオブジェクトのメタデータを保持する．我々のツールの文脈ではメタデータはイベント履歴（へのポインタ）である．拡張ポインタ表現に基づく関連付けの利点は，検査時に履歴を高速に取得できる点である[⋆2]．

その反面，拡張ポインタ表現は，検査コードと対象コードの互換性を維持できないという欠点を持つ．たとえば，対象コードがアセンブリコードやコンパイル済みの外部ライブラリ関数とポインタを介してデータをやりとりしている場合，拡張ポインタ表現から通常のポインタ表現への変換や逆の変換を行う必要がある[25),26)]．しかし，これらの変換を自動で完全に行うことは，現時点の技術では不可能である．したがって，手動による変換が要求される（互換性の損失）．

オブジェクト表に基づく検査コードでは手動によるポインタ表現の変換はまったく要求されない．我々は検査の高速性よりも互換性を重視し，拡張ポインタ表現ではなくオブジェクト表を用いてオブジェクトと履歴の関連付けを実現する．

3.2 遅 延 検 査

オブジェクト表に基づく検査コードは，対象コードと高度な互換性を維持できるため有望であるが，非同期割込みの処理中に競合状態を引き起こしてしまう．なぜなら，割込みハンドラに挿入された検査コードが非同期割込み安全でない関数を呼ぶからである．**図 5** の例

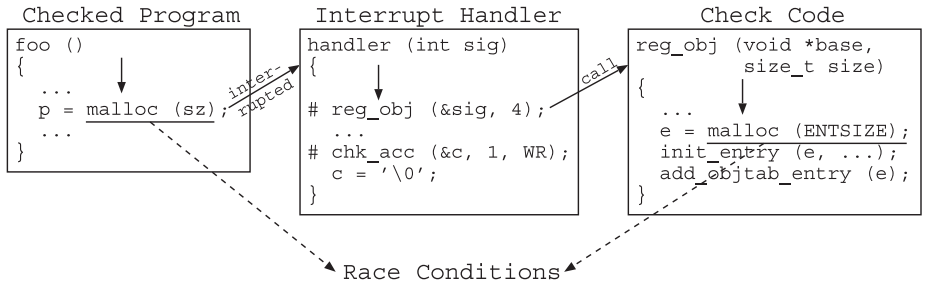

図 5　オブジェクト表に基づく検査コードが引き起こす競合状態
Fig. 5　Race Conditions caused by Object-Table-Based Check Code.

では，検査対象コードが関数 malloc の実行途中で割り込まれ，起動した割込みハンドラが検査コード（reg_obj）を実行し，検査コードが関数 malloc を呼び出してオブジェクト表のエントリを割り当てている．ここで，malloc は非同期割込み安全でないため，検査コード内の malloc の実行は競合状態を引き起こす危険性がある．同様に，割込みハンドラ内の他の検査コード（chk_acc）もイベント履歴のエントリの割当て/解放時に競合を引き起こしうる．さらに，検査コードが他の非同期割込み安全でない関数[⋆3]を実行する場合も同様である．

検査コードの競合状態に対する単純な回避策として，割込みハンドラへの検査コードの挿入を抑制することが考えられるが，我々の検出ツールでその方法は使用できない．なぜなら，挿入を抑制すると，割込み処理中のメモリアクセスを検査できなくなるからである．そこで，我々は遅延検査と呼ぶ検査手法を導入する．

遅延検査は割込み処理中の検査の実行を割込み終了後まで遅延させる．その結果，検査コードの競合回避と割込みハンドラのアクセス検査を両立させる．

遅延検査の実装の要点は以下のとおり：

- 検査対象プログラムの各プロセスの実行コンテキスト（割込み処理中か否か）を追跡管理する．
  この追跡管理は割込みディスパッチ（後述）によって実現する．
- 各プロセスが検査コード（検査関数 chk_acc，{reg,unreg}_obj の呼び出し）に到達したときに，実行コンテキストに応じて検査（検査関数の本体）の実行タイミングを制

⋆1 augmented pointer representation，あるいは，fat pointer.
⋆2 履歴は拡張ワードから即座に取得できる．表の探索などは不要．
⋆3 典型的には，検出報告に用いる入出力関数（fprintf など）やデバッグ情報操作用のライブラリ関数など．

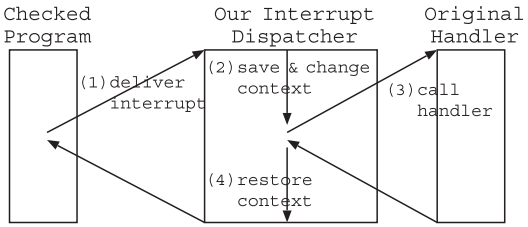

図 6　割込みディスパッチ
Fig. 6　Interrupt dispatching.

御する：

Case 1：コンテキストが “割込み中” の場合，検査の実行を割込み終了後まで遅延させる．

具体的には，検査関数は本体を実行せずに引数をプロセス固有のバッファ（検査バッファと呼ぶ）に保存してリターンする．

Case 2：“割込み中” でない場合，遅延中の検査を実行し，その後，現在の検査も実行する．

具体的には，検査関数は検査バッファに保存された一連の引数を適切な検査関数に与えて実行し，その後，自身の本体を実行する．

遅延検査により，2 つの効果が得られる．まず，**Case 1** の処理により，割込み処理中に検査コードが非同期割込み安全でない関数を呼ぶことを回避できる（検査コードの競合状態の回避）．次に，**Case 2** の処理により，割込みハンドラのメモリアクセスを割込み終了後に検査することができる（割込み競合の検査の実現）．

各プロセスの実行コンテキストの追跡管理は，割込みディスパッチで実現する．我々は各プロセスごとに個別のコンテキスト変数を割り当て，変数の値[⋆1]を以下の手順で管理する：

- **Step 1**：割込みが発生したとき，それを元の割込みハンドラではなく（検出ツールの）割込みディスパッチャに配送する（**図 6** (1)）．
- **Step 2**：ディスパッチャは現在のコンテキスト変数の値を保存した後，変数値を “割込み中” に変更する（図 6 (2)）．
- **Step 3**：ディスパッチャは受信した割込みに対応する元のハンドラを呼び出す（図 6 (3)）．
- **Step 4**：元のハンドラが完了した後，ディスパッチャはコンテキスト変数を元の値に復元し，リターンする（図 6 (4)）．

割込みディスパッチにより，各プロセスの実行コンテキストはコンテキスト変数に反映される．

最後に，割込みディスパッチの実装上の課題と我々の解決法を述べる．まず，上記の **Step 1** では，発生した割込みをディスパッチャに配送しなければならない．この課題の解決法として，我々は割込みハンドラ登録関数（`signal` や `sigaction` など）をラッパ関数に置換し，ラッパ関数内で元のハンドラの代わりにディスパッチャを登録する．

Step 3 を実行するには，各割込みと元のハンドラの対応関係を管理しなければならない．この課題の解決法として，我々はディスパッチ表を導入する．ディスパッチ表は割込み番号をインデクスとしハンドラ情報を要素とする固定長配列である．ハンドラ情報とはハンドラの実行に必要な情報であり，ハンドラへの関数ポインタやハンドラ実行中の割込みマスクなどを含む[⋆2]．前述のハンドラ登録関数のラッパは，ディスパッチャの登録前に，元のハンドラとハンドラ情報の対応関係をディスパッチ表に記録する．その結果，ディスパッチャは **Step 3** で，現在の割込みに対応する元のハンドラをディスパッチ表から取得できる．

Step 4 は元の割込みハンドラが `longjmp` や `siglongjmp` などで非局所脱出（non-local exit）を行う場合にスキップされてしまうため適切な対処が必要である．この課題の解決法として，我々は `longjmp` や `siglongjmp` をラッパ関数に置換し，ラッパ関数内で次の制御を行う：

- **Step 1**：脱出先のコードを最後に実行した際のコンテキストを取得する．
- **Step 2**：現在処理している割込みが同期割込みである場合に限り，取得したコンテキストの割込み状態と割込み番号 i をコンテキスト変数にリストアする．
- **Step 3**：元の関数（`longjmp` または `siglongjmp`）が脱出時のマスク復元を指定している場合に限り，取得したコンテキストの割込みマスク M をコンテキスト変数にリストアする．

この制御により，元の割込みハンドラが非局所脱出を行った場合でも，実行コンテキストを正しく復元することが可能となる．なお，上記の **Step 1** で必要となるコンテキストは，

⋆1 説明の便宜上，コンテキスト変数が保持する値として “割込み中” と “割込み中でない” だけを記す．ただし，実際の検出ツールの実装では，コンテキスト変数は，現在処理中の割込みの番号 i や現在の割込みマスク M などの情報も保持する．

⋆2 シグナルハンドラの場合，ハンドラ情報は `struct sigaction` の持つ情報である．

脱出先の設定時（`setjmp` や `sigsetjmp` の呼び出し時）に，ジャンプバッファに対応するオブジェクト表エントリに記録しておく．また，脱出先で発生する割込み競合（4 章を参照）を検出するために，**Step 2** では，処理中の割込みが非同期割込みである場合はコンテキスト変数の割込み状態と割込み番号 i を復元しない．すなわち，非同期割込みからの非局所脱出では，脱出先においても割込み処理中の扱いとする．

4. 実　　験

本章では，我々の競合検出手法の予備評価結果を示す．我々は提案手法に基づく検出器のプロトタイプ（TR2 と呼ぶ）を GCC 4.3.2 [23] の拡張として実装し，評価用プログラムに適用した．適用の目的は割込み競合の検出精度の計測である．実験環境は Intel Core2 Duo 1.33 GHz × 2 と 2 GB の RAM を搭載した Linux 2.6.24 ワークステーションである．コンパイル時の最適化レベルには-O2 を使用した．

表 1 が割込み競合の検出結果である．表中の列 Program は割込み競合を引き起こす実用 C プログラム名とそのバージョンを示す．ただし，我々は列 Program が示すプログラムそのものではなく，それらの割込み競合をモデル化した評価用プログラムを用いて実験を行った．モデルの作成法およびモデル上で競合を引き起こす方法については，Zalewski の手法[27] などを参考にした．表中の列 Vuln は該当の割込み競合に対して CERT [2] や MITRE [3] などの機関が割り当てた脆弱性番号を示す．列 TR2 は我々のプロトタイプによる競合検出の結果を示す．列 Crocus は Valgrind [28] の試験的ツール Crocus [9] による検出結果を示す．Crocus はシグナルハンドラ内での非同期シグナル非安全なライブラリ関数の呼び出しを潜在的競合として検出する．bash はシェルであり，sendmail は SMTP サーバであり，smail は MTA であり，wu-ftpd は FTP サーバである．

bash-3.0 では，2.4 節でも説明したとおり，ある共有変数をアクセスしている間に SIGHUP が発生し，対応するハンドラが同一の変数をアクセスしたときに割込み競合が発生する．我々の検出器はこの競合の検出に成功した．一方，Crocus は検出に失敗した．Crocus は共有変数上の割込み競合を検出することができないからである．

表 1　割込み競合の検出精度
Table 1　Accuracy of our interrupt-race detection.

Program	Vuln	TR2	Crocus
bash-3.0	-	yes	no
sendmail-8.11.3	CVE-2001-1349	yes	yes
sendmail-8.13.5	VU#834865	yes	no
smail-3.2.0.120	CVE-2005-0893	yes	yes
wu-ftpd-v2.4	-	yes	yes

sendmail-8.13.5 では，あるローカル変数をアクセスしている間に特定のシグナルが発生し，対応するハンドラ内で `longjmp` を呼び出し，ジャンプ先で同一の変数をアクセスしたときに割込み競合が発生する．列 TR2 が示すとおり，我々の検出器はこの競合を検出できた[⋆1]．一方，Crocus も競合の可能性を報告した．しかし，Crocus が競合の発生位置として報告したのは，シグナルハンドラ内で `longjmp` が呼び出される箇所であり，ジャンプ後にローカル変数がアクセスされる箇所ではなかった．したがって，Crocus の報告は誤検出である（競合の報告箇所が競合の発生箇所と一致しない）．同様に，非同期シグナルハンドラ内で `longjmp` を実行し，ジャンプ先で競合を引き起こさずに後処理を行うプログラムに対し，Crocus は誤検出を報告する．

sendmail-8.11.3，smail-3.2.0.120，wu-ftpd-v2.4 では，非同期シグナル安全でない関数の実行中にシグナルが発生し，対応するハンドラ内で同一の関数（または，その関数と内部状態を共有する関数）を実行したときに割込み競合が発生する．sendmail-8.11.3 は複数のハンドラ内で `free` を呼び出し，smail-3.2.0.120 は SIGHUP ハンドラ内で `malloc` を呼び出し，wu-ftpd-v2.4 は SIGURG ハンドラ内で `syslog` を呼び出し，これらの呼び出しが競合を引き起こす．3.1.2 項で説明したとおり，我々の手法は，非同期シグナル安全でない関数が内部で共有データに（非安全に）アクセスする過程を擬似アクセスとして生成し検査する．その結果，我々の検出器は上記の 3 つの競合をすべて検出できた．一方，Crocus もその仕様どおり，良い検出結果を示している．

以上，実用 C プログラムそのものではなく，それらの割込み競合のモデルに対する検査結果ではあるが，我々の競合検出法によって実用 C プログラムの割込み競合を高精度に検出可能であることが確認できた．

5. 関 連 研 究

データ競合の検出を目的として，現在までに多数の手法が提案されている．本章では，それらの中で実用 C プログラムを検査可能な手法に焦点を当て，我々の手法と比較する．

⋆1 該当のローカル変数に対するシグナル発生前のアクセスを e_1，`longjmp` 後のアクセスを e_2 とすると，特に $e_2.i \notin e_1.M$ であるため，検出器は $IsIntRace(e_1, e_2)$ の成立を検出する．

5.1　スレッド競合検出手法

既存のデータ競合検出法の大部分はスレッド競合の検出を目的としている．それらの検出法は動的手法と静的手法に大別できる．

動的手法は対象プログラムに検査コードを挿入し，プログラムを実行することでスレッド競合を検出する．動的手法は一般に，大規模なプログラムの検査を現実的な時間で完了できる反面，網羅的な検査を行うことが難しい．Eraser[5] はバイナリコードに検査コードを挿入し，各メモリ位置の状態をメモリアクセスの属性を入力とする単純なステートマシンで管理しつつ，lockset 解析を行う．Eraser は lockset 解析に基づく初期のスレッド競合検出ツールであり，後の動的検出手法[6),7),11),13),16)] に影響を与えた．Eraser はスレッドの `fork` 操作によるアクセス順序の強制には対応している．しかし，`join` 操作には未対応であるため，誤検出が発生する．Helgrind[11] は Valgrind[29] の付属ツールであり，Eraser と同様，バイナリコードを対象に検査を行う．Helgrind は対象プログラムを解釈実行しながら各メモリ位置の状態管理と lockset 解析を行いスレッド競合を検出する．Helgrind は，スレッドの `fork`/`join` 操作や条件変数を用いた同期によって強制されるアクセス順序を Lamport の happens-before 関係[30)] を用いて解析し，誤検出を低減させる．Muhlenfeld らの手法[13)] は Helgrind の改善であり，アノテーションと fake segment を用いて条件変数による同期に関する誤検出と検出漏れを低減させる．Jannesari らの手法[16)] は Helgrind の各メモリ位置の状態管理用ステートマシンを洗練して検出精度を向上させる．また，write/read 関係を用いて happens-before 解析をより正確に行うことで，条件変数に関する誤検出と検出漏れをさらに低減させる．

上記の手法はすべて，各メモリ位置を保護する lockset が一意に決まるという仮定に基づき，各メモリ位置に 1 個の lockset しか割り当てない．しかし，現実のプログラムでは，同一メモリ位置への複数のアクセスが異なる lockset で保護される場合がある．その場合に上記の手法は誤検出または検出漏れを引き起こす．

一方，Java プログラムを対象とする手法には，より高度な lockset 解析を行うものがある．Choi らの手法[6)] は各メモリ位置に対する各アクセスにつき 1 個の lockset を関連付けて記録する．そのため，各メモリ位置につき 1 個の lockset を仮定する手法に比べ，より高精度な検査が可能である．O'Callahan らの手法[7)] は Choi らの手法に happens-before 解析を導入し，検出精度を改善している．Elmas らの手法[31)] も happens-before 解析を行うが，この解析は洗練された lockset 解析（Goldilocks と呼ばれる）に基づく．Goldilocks は各メモリ位置に同期変数の集合やスレッド ID の集合などを関連付け，アクセスイベントの発生時や同期操作の実行時にこれらの集合を更新する．Godilocks では，他の lockset 解析と異なり，誤検出が発生しない．ただし，その反面，実装はより複雑である．

静的手法はソースコードを静的に解析し，スレッド競合が発生しうる箇所を特定する．静的手法は一般に，動的手法に比べて検査の網羅性に優れる反面，誤検出が多く，現実的な時間で検査を完了することが難しい．RacerX[8)] は flow-sensitive な手続き間解析を行って，各ソースコード位置での lockset を計算する．RacerX は各種の近似手法を用いることでスケーラビリティを向上させ，Linux[32)] などの 100 万行を超える大規模 C プログラムのスレッド競合を現実的な時間で検出した．しかし，それらの近似手法は多くの検出漏れを引き起こす．LOCKSMITH[10)] は correlation 解析と呼ぶ制約ベース解析により，各左辺値を保護する lockset を計算する．LOCKSMITH はすべての手続き間解析を単一のマシン上で行うため，複数モジュールにまたがるポインタエイリアスの伝播を比較的正確に追跡できる．その反面，スケーラビリティに課題があり，検査に成功したプログラムは最大で 2 万行程度と小規模である．RELAY[12)] も flow-sensitive な lockset 解析を行う．RELAY はプログラムの各モジュールを個別のマシン上で並列に解析することで，スケーラビリティの向上を達成し，100 万行を超えるプログラム（Linux）の検査に成功した．しかし，この手法は複数のモジュールを横断するポインタエイリアスを追跡できない．LP-Race[14)] はスレッド競合検出問題を線形計画問題に帰着させる．LP-Race は，条件変数を用いた同期や `join` 操作によるアクセス順序の強制も扱える反面，再帰ロックやスレッドを生成するループを処理できない場合がある．また，スケーラビリティの向上にも課題が残る．

以上の動的/静的手法はスレッド競合の検出を目的としており，割込み競合を検出することはできない．これに対し，我々の手法は割込み競合の検出を目的としており，Choi らの手法を改変することで従来手法より高精度な割込み競合の検出を実現している．検出精度の高さと実装の単純さのバランスを考慮し，我々は Choi らの手法を改変対象として選択した．

5.2　割込み競合検出手法

スレッド競合検出法に比べると数は非常に少ないが，割込み競合の検出を目的とする手法も存在する．

Crocus[9)] は，Valgrind[29)] の試験的ツールであり，シグナルハンドラ内での非同期シグナル安全でない（標準 C ライブラリ）関数の呼び出しを潜在的競合として検出する．4 章で示したとおり，Crocus はこれらの関数が引き起こす割込み競合の検出に有効であるが，シグナルハンドラとハンドラ外のコードによる同一変数へのアクセス競合を検出できない．DRACULA[15)] は，シグナルハンドラとハンドラ外のコードによる同一のグローバル変数

へのアクセス競合を検出する．DRACULA は検査対象プログラムのプロセスをデバッギとして実行し，各グローバル変数へのアクセスをウォッチポイントで捕捉する．捕捉直後に，DRACULA はデバッギに全種類のシグナルを送信し，起動したハンドラが同一変数にアクセスした際に競合を報告する．DRACULA はグローバル変数上のアクセス競合を（シグナル送信により）意図的に発生させることで検出精度の向上を図っている．しかし，ルートプロセス以外のプロセスを検査できない，グローバル変数以外のメモリオブジェクトへのアクセス競合を検出できないなど，検出精度上，重大な問題点をかかえる．

以上の割込み競合検出法に比べ，我々の手法はより高精度な検査が可能である．我々の検出器はあらゆる共有オブジェクトへのアクセス競合を検出できるため，Crocus より高精度である．また，我々の手法では対象プログラムのすべてのプロセスが検査対象であり，共有オブジェクトがグローバル変数でない場合であっても競合を検出できる．これらの特徴が DRACULA に対する優位性である．

一方，我々の手法は検査コードを挿入するために，対象プログラムのソースコードと再コンパイルを必要とする．これに対し，Crocus と DRACULA はバイナリコードを用いて検査を行うため，それらの必要はない．この特徴は，検出器の適用コストの観点から，我々の手法より優れる．

6. 結論と今後の展望

本稿では，実用 C プログラムの割込み競合を高精度に検出する動的手法を提案した．我々は割込み処理を擬似的なスレッド実行として扱い，割込み競合の検出問題を Choi らのスレッド競合の検出問題に帰着させた．その結果，従来手法より高精度な割込み競合の検出が可能となった．また，オブジェクト表（および，遅延検査）に基づく検査コードの実装を示した．この実装により，対象プログラムと検査コードの互換性を高度に維持することが可能となった．提案手法に基づく検出ツールを用いた実験では，Sendmail や WU-FTPD を含む実用 C プログラムの割込み競合を高精度に検出できることが確認できた．以上より，我々の提案手法は，実用 C プログラムの割込み競合の高精度な検出に有効であるといえる．

今後の展望として，2.5 節で述べた個々の課題を（部分的に）解決することを計画している．

謝辞　本研究は，ルネサステクノロジ，日立製作所，早稲田大学，東京工業大学の共同プロジェクトである NEDO （New Energy and Industrial Technology Development Organization）P05020 から一部支援を受けました．

参 考 文 献

1) Netzer, R.H.B. and Miller, B.P.: What Are Race Conditions?: Some Issues and Formalizations, *ACM Lett. Program. Lang. Syst.*, Vol.1, No.1, pp.74–88 (1992).
2) CERT/CC. http://www.cert.org/advisories/
3) MITRE. http://www.mitre.org/
4) SecurityFocus. http://online.securityfocus.com/
5) Savage, S., Burrows, M., Nelson, G., Sobalvarro, P. and Anderson, T.E.: Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs, *ACM Trans. Comput. Syst.*, Vol.15, No.4, pp.391–411 (1997).
6) Choi, J.-D., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V. and Sridharan, M.: Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs, *Proc. 2002 ACM SIGPLAN Conference on Programming Languages Design and Implementation* (*PLDI'02*), New York, NY, USA, pp.258–269, ACM Press (2002).
7) O'Callahan, R. and Choi, J.-D.: Hybrid Dynamic Data Race Detection, *Proc. 2003 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (*PPoPP'03*), New York, NY, USA, pp.167–178, ACM Press (2003).
8) Engler, D. and Ashcraft, K.: RacerX: Effective, Static Detection of Race Conditions and Deadlocks, *Proc. 2003 ACM Symposium on Operating Systems Principles* (*SOSP'03*), New York, NY, USA, pp.237–252, ACM Press (2003).
9) Valgrind-project.: Crocus: A signal-handler checker (2005). http://valgrind.org/downloads/variants.html?njn
10) Pratikakis, P., Foster, J.S. and Hicks, M.: LOCKSMITH: Context-Sensitive Correlation Analysis for Race Detection, *Proc. 2006 ACM SIGPLAN Conference on Programming Languages Design and Implementation* (*PLDI'06*), New York, NY, USA, pp.320–331, ACM Press (2006).
11) Valgrind-project.: Helgrind: A data-race detector (2007). http://valgrind.org/docs/manual/hgmanual
12) Voung, J.W., Jhala, R. and Lerner, S.: RELAY: Static Race Detection on Millions of Lines of Code, *Proc. 2007 Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (*ESEC-FSE'07*), New York, NY, USA, pp.205–214, ACM Press (2007).
13) Muhlenfeld, A. and Wotawa, F.: Fault Detection in MultiThreaded C++ Server Applications, *Proc. 2007 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (*PPoPP'07*), New York, NY, USA, pp.142–153, ACM Press (2007).

14) Terauchi, T.: Checking Race Freedom via Linear Programming, *Proc. 2008 ACM SIGPLAN Conference on Programming Languages Design and Implementation* (*PLDI'08*), New York, NY, USA, pp.1–10, ACM Press (2008).
15) Tahara, T., Gondow, K. and Ohsuga, S.: DRACULA: Detector of Data Races in Signals Handlers, *Proc. 2008 Asia-Pacific Software Engineering Conference* (*APSEC'08*), Los Alamitos, CA, USA, pp.17–24, IEEE Computer Society (2008).
16) Jannesari, A., Bao, K., Pankratius, V. and Tichy, W.F.: Helgrind+: An Efficient Dynamic Race Detector, *Proc. 2009 IEEE International Parallel and Distributed Processing Symposium* (*IPDPS'09*), Los Alamitos, CA, USA, pp.1–13, IEEE Computer Society (2009).
17) The Sendmail Consortium. http://www.sendmail.org/
18) The WU-FTPD Project. http://www.wu-ftpd.org/
19) BASH (GNU Project). http://www.gnu.org/software/bash/
20) US-CERT Vulnerability Note VU#834865. http://www.kb.cert.org/vuls/id/834865
21) Common Vulnerability and Exposures CVE-2001-1349. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2001-1349
22) Common Vulnerability and Exposures CVE-1999-0035. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-1999-0035
23) Free Software Foundation (FSF): GCC, the GNU Compiler Collection. http://gcc.gnu.org/
24) Sleator, D. and Tarjan, R.: Self-adjusting binary search trees, *J. ACM*, Vol.32, No.3, pp.652–686 (1985).
25) Ruwase, O. and Lam, M.: A Practical Dynamic Buffer Overflow Detector, *Proc. 11th Annual Network and Distributed System Security Symposium*, San Diego, California, pp.159–169 (2004).
26) Dhurjati, D. and Adve, V.: Backwards-Compatible Array Bounds Checking for C with Very Low Overhead, *Proc. 2006 International Conference on Software Engineering* (*ICSE'06*), Shanghai, China (2006).
27) Zalewski, M.: *Delivering Signals for Fun and Profit*, BindView Corporation (2001).
28) Nethercote, N. and Seward, J.: Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation, *Proc. 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI'07*), San Diego, California, pp.89–100 (2007).
29) Nethercote, N. and Seward, J.: Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation, *Proc. 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI'07*), San Diego, California, pp.89–100 (2007).
30) Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System, *Comm. ACM*, Vol.21, No.7, pp.558–565 (1978).
31) Elmas, T., Qadeer, S. and Tasiran, S.: Goldilocks: A Race and Transaction-Aware Java Runtime, *Proc. 2007 ACM SIGPLAN Conference on Programming Languages Design and Implementation* (*PLDI'07*), New York, NY, USA, pp.245–255, ACM Press (2007).
32) The Linux Kernel Project: The Linux Kernel Archives. http://www.kernel.org/

(平成 22 年 1 月 7 日受付)
(平成 22 年 6 月 3 日採録)

荒堀　喜貴

2010 年東京工業大学大学院情報理工学研究科博士課程計算工学専攻修了．同年同大学院情報理工学研究科計算工学専攻特別研究員．博士（工学）．ソフトウェア開発環境・システムプログラミングに興味を持つ．

権藤　克彦

1994 年東京工業大学大学院理工学研究科博士課程情報工学専攻修了．同年同大学院情報理工学研究科情報工学専攻助手，講師を経て，1998 年より北陸先端科学技術大学院大学助教授．ブラウン大学客員研究員（2000〜2001 年）．2003 年より東京工業大学助教授（現在は同准教授）．博士（工学）．ソフトウェア開発環境・システムプログラミングに興味を持つ．著書『例解 UNIX プログラミング教室』『Java によるプログラミング入門』．ACM，日本ソフトウェア科学会，電子情報通信学会各会員．

前島　英雄（正会員）

1973 年東京工業大学大学院理工学研究科修士課程制御工学専攻修了．同年（株）日立製作所入社，部長，主管研究員を経て，1999 年より東京工業大学大学院総合理工学研究科教授．工学博士．マイクロプロセッサ，特にマルチコアやリコンフィガラブル・アーキテクチャに興味を持ち，最近はソフトウェア統合開発環境の研究も行っている．IEEE 会員，電子情報通信学会フェロー．