RogueWave SOFTWARE

TotalView 操作ガイド

1. プログラムをコンパイル

デバッグ対象のプログラムを **-g** オプションを付けてコンパイルして下さい。

例： ***gcc*** *–g* *–o* ***<my_prog>*** ***<my_prog>.c***

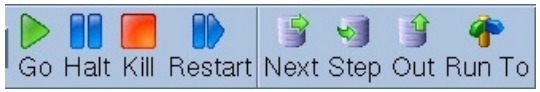
2. TotalView を開始

TotalView 起動例： TotalView ***<my_prog>*** –a 引数

または"TotalViewとだけ入力して**「**New Program dialog box**」**の下記項目からデバッグするプログラムを指定

- Start a new process
- Attach to running process
- Open a core file

3. TotalView のボタンについて



- **Go:** 実行開始
- **Halt:** 一時停止（リスタート可）
- **Kill:** 実行終了
- **Restart:** 実行終了 & 実行開始
- **Next**: 現在の行を実行。プログラムカウンタ（PC）は次の行に
- **Step:** 現在の行を実行。関数実行があれば PC はその中へ移動
- **Out:** 現在の関数の残りを実行し、呼び出し行の次の行に PC を移動
- **Run To:** PC から選択した行（行はグレイアウトする）までを実行

4. ブレークポイントの設定方法

- **行：**行番号をクリック
- **関数：**「Action Point」→「At Location」を選択、関数名を入力
- **関数：**「View」→「Lookup Function」を選択、行番号をクリック

5. 実行中のプログラムにアタッチ

- **「File」→ 「New Program」**を選択し、**「Attached to process」**ボタンをクリック。プログラム名をクリックして、**「OK」**ボタンを押す。
- プログラムが見えないなら、**ps** コマンドで PID を得て、**「File」→ 「New Program」**ダイアログボックスでその PID を選択。

必ずプログラムのメインスレッドにアタッチしてください。

6. 変数がある値に等しい時（または等しくない時）にある行で実行をストップ

a. ループ中にブレークポイントを設定
b. ブレークポイントアイコンを右クリックし、「Properties」を選択
c. ダイアログボックスの「Evaluate」を選択
d. 条件を入力

例：`if (my_variable == 0) $stop`

7. イテレーション(繰り返し実行)を数えて実行をストップ

a. 前の項目の a から c まで実行
b. イテレーション 100 回毎にストップするには:

`$count 100`

8. 変数の値を見る

- ローカル変数の場合、「Stack Frame」ペインを参照。ローカル変数もしくはグローバル変数の場合、「Source」ペインを参照。それぞれ変数名をダブルクリック、もしくは右クリック→「Dive」で Variable Window に値が表示される。またはマウスを変数上に移動するとツールチップに変数の値が表示される。
- 単純な配列や構造体の場合は、変数を右クリックして、「**Add to Expression List**」を選択
- 配列や構造体をダブルクリックすると Variable Window に値が全て表示

9. ポインタ値を追跡

変数のタイプがポインタの場合は、ダブルクリックで値を表示

10. 沢山の変数を同時に見る

「Expression List」ウィンドウには複数の変数を表示できます。これらの値はプログラムの実行が休止される時更新されます。個別の構造体や配列要素をこのウィンドウに送ることも可能です。

11. 配列要素の一部だけを見る

Variable Window の**「Slice」**エリアは、配列要素のどの部分を表示するか指定できます。例えば Fortran で(31:60)、C や C++で[30:59]と入力すると、30 個の要素のみ表示します。

特定の範囲内に表示を制限するには、**「Filter」**エリアに条件を入力して下さい。例えば **「> 64000」**の入力は 64000 より大きな値の要素のみ表示します。「Slice」と「Filter」は組み合わせて使うこともできます。

12. 配列をグラフ表示

Variable Window で**「Tool」→「Visualize」**を選択して配列データを可視化することによって、外れ値やデータパターンを容易に検出できます。

13. キャスト

Variable Window の「Type」フィールドを編集して、変数データを表示する型を変えることができます。例えば、配列を指すポインタのデータ型を **int *** から **int[100]**に変更することによって、配列の要素を見ることができます。

14. 変数の値の変更

- 「Expression List」や Variable Window で、値をクリックして変更します。
- 「Stack Frame」ペインで、太字の値をクリックして変更します。

15. STL 変数 (STLView)

TotalView は自動的な STL Type Transformations を提供しています。Preferences からオン/オフが可能です。

16. 変数のサーチ

プロセスウィンドウから、**「View」→「Lookup Variable」**でサーチした変数が Variable Window に表示されます。

17. 変数の値が変更された時に実行を中断 (ウォッチポイント)

変数上で右クリック→**「Create Watchpoint」**。Variable Window に表示されている変数が配列や構造体の場合は、要素が 1 つだけ表示されるようにダイブして下さい。

18. 構造体の配列の一要素をその変数の配列であるかのように表示

a. 要素を 1 つ選択する
b. 右クリック→**「Dive in All」**を選択する

ウィンドウはその要素だけから成る配列を表示します。

19. マルチスレッドやマルチプロセスの変数を見る

Variable Window のメニューから選択

- マルチスレッドの時は**「Views」→「Show Across」→「Thread」**
- マルチプロセスの時は**「Views」→「Show Across」→「Process」**

「Stack Frame」または「Source」ペインから、変数を右クリックして**「Across Processes」**または**「Across Threads」**を選択

20. CLI コマンドエントリ

「Tools」→「Command Line」を選択。このウィンドウから TotalView CLI コマンドを入力。例えばステップ実行なら **dstep** (略字で **s**)、ヘルプは **dhelp** と入力。CLI コマンドは Tcl のインタープリタに組み込んだりスクリプトに書くことも可能

21. fork()及び Execve()プログラムをデバッグ

多くの場合、プログラムは TotalView が提供する libdbfork ライブラリとリンクする必要があります。詳しくは Reference ガイドを参照してください。

22. ReplayEngine を使用したデバッグ

ReplayEngine は Linux x86, x86-64 で利用可能な TotalView アドオンで、リバースデバッギングを行います。ReplayEngine を使用する時は、デバッギングセッションの前に以下のいずれかを行って下さい。

- 「New Program」ダイアログボックスで**「Enable ReplayEngine」**を選択
- プロセスウィンドウのメニューから**「Debug」→「Enable ReplayEngine」**を選択

ツールバー上の ReplayEngine のボタンは次の通り：

- **GoBack:** 最も近いストップイベントまで逆向きに実行する
- **Prev:** 逆向きに 1 行実行。関数はまたぐ。PC(プログラムカウンタ)は 1 行前に
- **UnStep:** 逆向きに 1 行実行。PC は関数の中に動く
- **Caller:** PC は現在の関数の呼び出し元に戻る
- **Back To:** 選択された行（その行はグレイアウト）まで逆向き実行
- **Live:** PC はカレントの live 行（実行された最も後ろの行）に戻る
- **Save:** 記録されたリプレイセッションをファイルに保存し、後でロードできるようにする

RogueWave SOFTWARE

MemoryScape 操作ガイド

1. MemoryScape の起動

プログラムを実行する前に起動する。起動方法は 3 通り

- New Program ウィンドウで、 **Enable Memory Debugging** を選択
- Process ウィンドウで **Debug > Enable Memory Debugging** を選択
- コマンドラインで、memscape と入力する（この場合 MemoryScape は TotalView なしで起動）

2. メモリエラーをチェック

メモリデバッグを有効にした場合、MemoryScape は以下の様な問題が発生するとプログラム実行を停止ししてイベントフラグを立てます。

- stack、bss、data セクションのメモリ解放
- 一度解放したメモリを再度解放
- 誤ったアドレスで解放
- 内部ポインタを解放
- ブロックの不正なアラインメント
- 未知のブロックを再アロケーション
- 不正なアクセスを RedZone が検知

3. バックトレースとは

プログラムがメモリのリクエストを行ったら、MemoryScape はそのアクションが派生した時のスタックフレームを記録します。このフレームのリストをバックトレース(backtrace)と呼びます。

バックトレースを精査すると、そのリクエストが発生した命令文までにどのような実行パスを経由したのかが分かります。つまり、全てのメモリ確保のリクエストは最終的には malloc()や free()に行き着くため、知るべきはどのようにそこへたどり着くかということです。

4. メモリイベント情報を表示

TotalView のイベントウィンドウや MemoryScape のイベントフラグをクリックするとイベントの詳細情報が表示されます。メモリリクエストはいろいろなやり方で発生するため、MemoryScape はバックトレースと実行文を組み合わせることでこの情報を特定します。スタックフレームをクリックすると対応するソースコードが表示されます。イベントウィンドウの他のタブではメモリブロックが割り当てられたり解放された場所を確認できます。Block Details タブではメモリブロックの内容を確認できます。

5. メモリリークを検出/表示

ブレークポイントや **Halt** ボタンなどでプログラムの実行を停止します。停止ごとにプログラムのリークに関するレポートを要求できます。

何も見えなかったらそれは良いニュース。コードにメモリリークはありません。リークがあったら MemoryScape はリークの数やメモリの量を backtrace や確保時のソースコードと関連付けて集計します。これにより重大なリークに集中できます。

6. メモリ破壊の検出

バッファオーバーランなどのメモリの不正アクセスは Guard blocks や Red Zones 機能で検出可能です。

Guard blocks：メモリブロックの境界を越えて行われた書き込みを検出します。Guard blocks をオンにするには、「Basic Memory Debugging Options」から **Medium** を選択するか、「Advanced Memory Debugging Options」から「**Guard allocated memory**」を選択します。違反はメモリ解放時にチェックされ、通知されます。

Red Zones：読み/書き両方のメモリアクセス違反を検出し、プログラムが割り当てたブロックの範囲を超えた場合は即時にプログラムの実行を停止してイベントを通知します。オンにするには「Basic Memory Debugging Options」で **High** を選択するか「Advanced Memory Debugging Options」で「**Use Red Zones to find memory access violations**」を選択します。

7. ヒープをグラフィカルに可視化

ヒープメモリをグラフで可視化して検証すればプログラムがどのようにメモリを使っているのか多くの知見が得られます。プログラムのメモリ使用状態を分析するには「Memory Report」ページの「Heap Graphical Report」を選択します。

MemoryScape ウィンドウ上部でブロックを選択すると、そのブロックの情報が下部に表示されます。同じバックトレースに関連づけられた割り当ての数と、同じ場所から割り当てられたメモリの量が表示されることも重要です。「Heap Status Graphical Reports」ページのその他のレポートでは、割り当てに関連するバックトレースおよびソース行を表示できます。

8. 情報をフィルタリング

MemoryScape が表示する情報は膨大で、時にはメモリ割り当ての表示数が、数千～数万にもなることもあります。これらをプロセスやライブラリ、ソースファイルやクラス名、行番号、PC やなどでフィルタリングすることによって表示を簡易化できます。例えばサードパーティのライブラリや自分のコードの一部に興味がない場合には表示をオフにすることができます。

フィルターの追加や編集はレポート画面から右クリックで「Filter out this entry」を選択し、**Memory Debugging Data Filters** ダイアログを表示させます。

9. ダングリングポインタを特定

ダングリングポインタは、すでに解放されたメモリを指し続けているポインタです。メモリデバッグが有効である場合、追加で何かを行う必要はありません。TotalView の Variable Window や Stack Frame ペインで変数の値を表示したら、メモリが確保/解放されたか、それともダングリングポインタなのか分かります。

10. メモリブロックの詳細情報

MemoryScape の多くの場所から右クリックで Block Properties ウィンドウを開くことができ、メモリブロックについての様々な情報を参照できます。確保時/解放時のソース、始点と終点のアドレス、バイトの内容、文字列情報、バックトレースなど。コメントエリアにメモを入れればなぜそのブロックを調査したいのか思い出すことができます。

11. メモリの使い方をトラッキング

プログラムがどのくらいのメモリを使ったのか、は **Memory | Memory Usage** からわかります。Text/Data/Heap/Stack/仮想メモリ。プロセスごと、ソースファイルごと、ライブラリごとの使用量などの詳細情報、比較しやすいチャート表示。他のデータと異なり情報は OS から取得するため、情報を得るために事前にメモリデバッグをオンにしておく必要はありません。

12. メモリのペインティング

Block Painting は確保したがまだ初期化していないメモリや解放済みのメモリにアクセスすることによって生じる問題を特定するのに役立ちます。

Block Painting は新しく確保されたブロックや解放されたブロックに特定のパターンを書き込みます(デフォルトでは確保用は 0xa110ca7f です。allocate に似ているでしょう？)。もしそのパターンを目にすることがあったら、それは何か問題が存在している、ということになります。

このペインティング機能を使うには、**Memory Debugging Options** から **Extreme** を選び、必要なら描き込みパターンを指定します。

13. メモリの解放/再利用を通知

- **Memory Reports | Heap Status | Heap Status Graphical Report** 画面から通知の必要なブロック上で右クリックし、Properties を選択。
- **Hide Backtrace Information** を選択し、+をクリックしてブロックを拡大。
- ウィンドウの最下部で **Notify when deallocated** を選択。

これでこのブロックのメモリが解放されたら TotalView は実行を止め、**Memory Event Details** ウィンドウを表示します。

14. ベースラインからのメモリの使用をトラッキング

- TotalView の Process Window から **Debug > Heap Baseline > Set Heap Baseline** を選択すると、MemoryScape が現在のメモリ状態を記録します。
- プログラムを実行し、停止したら、**Heap Change Summary** を選択。MemoryScape を使ってベースラインの設定以降のメモリ確保やリークを調査することができます。
- MemoryScape 内の一部のレポートにも「**Relative to baseline**」ボタンがあり、ベースライン設定後に生じた割り当てとリークだけを表示できます。

15. メモリのデータをファイル保存

- **Export Memory Data** コマンドでメモリの情報をファイルに保存することができます。マルチプロセスの場合、プロセス番号が自動的に付加されます。
- 後から **Home | Add Program** スクリーンから **Add Memory Debugging file** コマンドで保存しておいた情報を MemoryScape に取り込み、普段と全く同じ方法で調査できます。
- メモリのレポートを HTML ファイルに出力することもできます。

16. メモリの使い方を比較

- **Home | Summary** 画面のグラフで総ヒープ量をリアルタイムで監視できます。
- **Memory Reports | Memory Comparison** 画面で現在のプログラムのメモリ使用状況と、ファイルから読み込んだ以前の使用状況、そして 2 つの差分を並べて表示します。
- マルチプロセスのプログラムでは、プロセス同士の使用状況を比較することもできます。これによりリソースの分散が予定通りかどうかを検証できます。

17. 解放されたメモリ領域を Hoarding

- めったに使われない機能で、解放された領域をプログラムがそれと知らずにアクセスしてしまう際に発生する問題を検出します。
- 仕組みはシンプルで、解放リクエストを実行せず実際にはその領域を貯めこんで(Hoarding)監視対象とします。そのためこの機能は大量のメモリを消費する可能性があります。
- **Memory Debugging Options** 画面で Extreme を選択します。そして使用するメモリサイズを選択します。メモリ不足に注意してください。