目次

1 デバッガ 2
2 GDB 3
2.1 GDB の起動 3
2.1.1 ファイルオプション 3
2.1.2 モードオプション 4
2.1.3 起動に GDB が行う動作 5
2.1.4 初期化ファイル 6
2.1.4.1 ホームディレクトリの初期初期化ファイル 6
2.1.4.2 システム全体の初期化ファイル 7
2.1.4.3 ホームディレクトリ初期化ファイル 7
2.1.4.4 ローカルディレクトリ初期化ファイル 7
2.2 GDB の終了 7
2.3 シェルコマンド 7
2.4 ロギング出力 7
3 GDB コマンド 8
3.1 コマンド構文 8
3.2 コマンド設定 8
3.3 コマンド補完 8
3.4 ファイル名引数 8
3.5 コマンドオプション 9
3.6 ヘルプ 9
4 GDB でプログラムを実行する 10
4.1 デバッグのためのコンパイル 10
4.2 プログラムの開始 10
4.3 プログラムの引数 11
4.4 プログラムの環境 11
4.5 プログラムの作業ディレクトリ 11
4.6 プログラムの入出力 12
4.7 すでに実行中のプロセスのデバッグ 12
4.8 子プロセスの終了 12
4.9 複数の下位接続とプログラムのデバッグ 13
4.9.1 inferior 固有のブレークポイント 13
4.10 複数スレッドのプログラムのデバッグ 13
4.11 フォークのデバッグ 14
4.12 チェックポイント、再起動 15
5 停止と継続 15
5.1 ブレークポイント、ウォッチポイント、キャッチポイント 15
5.1.1 ブレークポイントの設定 15
5.1.2 ウォッチポイントの設定 16
5.1.3 キャッチポイントの設定 17
5.1.4 ブレークポイントの削除 17

5.1.5 ブレークポイントを無効にする 17
5.1.6 ブレーク条件 18
5.1.7 ブレークポイントコマンドリスト 18
5.1.8 動的 printf 18
5.1.9 ブレークポイントをファイルに保存する方法 19
5.1.10 静的プローブポイントの一覧表示 19
5.1.11 ブレークポイントを挿入できません。 19
5.2 継続とステップ 19
5.3 関数とファイルのスキップ 20
5.4 シグナル 21
5.5 スレッドストップ 22
5.5.1 オールストップモード 22
5.5.2 ノンストップモード 22
5.5.3 プログラムを非同期実行する 22
5.5.4 スレッド固有のブレークポイント 23
5.5.5 システムコール割り込み 23
5.5.6 オブザーバモード 23
6 後ろ向きのプログラム実行 24
7 Inferior の実行の記録と再生 24
8 スタックの検証 26
8.1 スタックフレーム 26
8.2 バックトレース 26
8.3 フレームの選択 26
8.4 フレーム情報 27
8.5 それぞれのフレームにコマンドを適用する 27
9 ソースファイルの検証 27
9.1 リスト 27
9.2 位置指定 28
9.3 編集 28
9.4 検索 28
9.5 ソースパス 28
9.6 機械語 28
9.7 ソース読み込み無効化 28

1 デバッガ

プログラムのバグ(bug)を取り除く(de-)ことをデバッグといいます。デバッグを行う手法はいくつかあり、例えばプログラム中に標準出力を行う命令を追加してデバッグを行う print デバッガと呼ばれる方法があります。デバッガはデバッグを支援するツールで、プログラムの任意箇所での停止や、変数の値の表示や変更、スタックトレースやメモリ内容の監視など高度な機能によりデバッグを支援します。

C 言語で書かれたプログラムに対応するデバッガはいくつか存在しており、有名なものに GDB と LLDB が存在します。このドキュメントではこの二つのデバッガについて基本的な使用方法の解説を行います。

2 GDB

GDB は Gnu Project のデバッガです。

2.1 GDB の起動

GDB を起動するには以下のいづれかのコマンドを使用します。起動後はコマンドを受け付けます。

```
gdb [options] [executable-file [core-file or process-id]]
gfb [options] --args <executable-file> [inferior-arguments ...]
```

`--args` を指定する場合、実行可能ファイルの後の引数(inferior-arguments) が実行時に渡されます。例えば `gdb --args gcc -O2 -c foo.c` は `gcc -O2 -c foo.c` の実行にデバッガをアタッチします。

options に指定できるオプションは `gdb -h` で確認できます。

2.1.1 ファイルオプション

GDB が起動すると、options 以外の引数は実行ファイルとコアファイル(またはプロセス ID)を指定するものとして読まれます。つまり`-se`、`-c` としてです。

`-symbols <file>, -s <file>`

file からシンボルテーブルを読み取ります。

`-exec <file>, -e <file>`

file を実行ファイルとして読み込みます。

`-se <file>`

file からシンボルテーブルを読み取り、実行ファイルとして使用します。

`-core <file>, -c <file>`

file をコアダンプとして検査します。

`-pid <number>, -p <number>`

プロセス ID が number のプロセスにアタッチします。

`-command <file>, -x <file>`

file からコマンドを実行します。

`-eval-command <command>, -ex <command>:`

単一の GDB コマンドを実行します。複数回指定可能です。

`-init-command <file>, -ix <file>`

下位ファイルをロードする前かつ gdbinit ロード後に file からコマンドを実行します。

`-init-eval-command <command>, -iex <command>`

下位ファイルをロードする前かる gdbinit ロード後に GDB コマンド `command` を実行します。

-early-init-command <file>, -eix <file>

出力生成前にファイルからコマンドを実行します。

-early-init-eval-command <command>, -eiex <command>

出力生成前に GDB コマンド command を実行します。

-directory <directory>, -d <directory>

directory をソースファイルとスクリプトファイルを検索するパスに追加します。

-readnow, -r

各シンボルファイルのシンボルテーブル全体を起動時に読み取ります。デフォルトではこの機能はオフになっています。

--readnever

各シンボルファイルのシンボルテーブルを読み取らないようにします。このオプションをつけるとシンボリックデバッグが実行できなくなります。

2.1.2 モードオプション

GDB はさまざまなモードで実行できます。

-nx, -n

初期化ファイルにあるコマンドを実行しません。

-nh

ホームディレクトリ初期化ファイルにあるコマンドを実行しません。システム全体及びカレントディレクトリの初期化ファイルは実行されます。

-quiet, -silent, -q

起動時のメッセージを表示しません。これらメッセージはバッチモードでも表示されません。コマンドによりこのオプションを有向化することもできます。

-batch

バッチモードで実行します。-x で指定したコマンドファイルのコマンドがすべて実行された後、終了コード 0 を返して終了します(-n が指定されていない場合は初期化ファイルのコマンドも実行されます)。ファイル内のコマンド実行中にエラーが発生した場合は 0 以外のステータスコードを返して終了します。

-batch-silent

バッチモードで実行し、かつ全く標準出力への出力を行いません。

-return-child-result

GDB の終了ステータスをデバッグ中のプロセスの終了コードにします。ただし (1)GDB が異常終了した場合、(2)ユーザが明示的に終了ステータスを指定した場合、(3)子プロセスが実行されないか終了しない場合(終了ステータスは-1 になる)の三つの場合を除きます。

-nowindows, -nw

GDB に GUI インターフェースがある場合、CUI のみを使用するように指定します。

windows, -w

GDB に GUI インターフェースがある場合、GUI インターフェースを使用します。

-cd <directory>

作業ディレクトリを directory に移動して実行します。

-data-directory <directory>, -D <directory>

directory をデータディレクトリ(GDB が補助ファイルを検索する場所)として実行します。

-fullname, -f

スタックフレーム表示時およびプロセス停止時に完全なファイル名と行番号を出力します。

-annotate <level>

GDB 内の注釈レベルを設定します。これはプロンプト、式の値、ソース行、その他の出力とともに GDB が出力する情報の量を制御します。レベル 0 が通常、1 が Gnu Emacs で使用され非推奨、レベル 3 は最大の注釈です。

--args

実行ファイル以降の引数をすべて下位のコマンドライン引数として渡します。

--baud <bps>, -b <bps>

GDB がリモートデバッグに使用するシリアルインターフェイスの回線速度を設定します。

-l <timeout>

GDB がリモートデバッグに使用する通信のタイムアウト(単位:秒)を設定します。

-tty <device>, -t <device>

プログラムの標準入力と出力に device を使用して実行します。

-tui

TUI(Text User Interfaec) モードをアクティブにします。TUI はターミナル上の複数のテキストウィンドウを管理し、ソース、アセンブリ、レジスタ、およびコマンド出力を表示します。

-interpreter <interp>

制御プログラムまたはデバイスとのインターフェイスにインタープリター interp を使用します。このオプションは GDB をバックエンドとして GDB と津伸によって設定されることを目的としています。

-write

実行ファイルとコアファイルを読み取り書き取りの両方で開きます。

-statistics

GDB は各コマンドを完了してプロンプトに戻った後、時間とメモリ使用量に関する統計情報を表示します。

-configuration

GDB はビルド時の構成パラメータの詳細を出力し、終了します。

2.1.3 起動に GDB が行う動作

セッション起動時に GDB が行う処理を以下に示します。

1. 基本的な内部状態を初期化します。
2. ホームディレクトリにある初期初期化ファイルが存在する場合、コマンドを読み取ります。
3. `-eiex` と `-eix` で指定されたコマンドとコマンドファイルを指定された順番に実行します。
4. コマンドラインで指定されたコマンドインタープリターを設定します。
5. システム全体の初期化ファイルと初期化ディレクトリからファイルを読み取ります。
6. ホームディレクトリ内の初期化ファイルを読み取り、ファイル内のすべてのコマンドを実行します。
7. `-iex` および `-ix` で指定されたコマンドとコマンドファイルを指定された順番に実行します。通常 `-ex` および `-x` を代わりに使用します。この方法では GDB 初期化ファイルが実行される前および inferior がロードされる前に設定を適用できます。
8. コマンドラインオプションとオペランドを処理します。
9. 現在の作業ディレクトリにある初期化ファイルを読み込んで実行します。
10. デバッグするまたはアタッチするプログラムまたはコアファイルが指定されている場合、GDB はプログラムまたはそのロードされた共有ライブラリ用に提供された自動ロードスクリプトをロードします。
11. `-ex` および `-x` で指定されたコマンド及びコマンドファイルを読み込んで実行します。
12. *history file* に記録されたコマンド履歴を読み取ります。

2.1.4 初期化ファイル

GDB 起動時に GDB はいくつかの初期化ファイルからコマンドを実行します。これらの初期化ファイルはコマンドファイルと同じ構文を使用し、同様に処理されます。

起動時にロードされる初期化ファイルのリストをロードされる順番で表示するには `gdb --help` が使用できます。

初期初期化ファイルは初期化プロセスの非常に速い段階でロードされます。ここでは `set` または `source` コマンドのみを配置できます。

ほかの一般の初期化ファイルは任意のコマンドを実行できます。

2.1.4.1 ホームディレクトリの初期初期化ファイル

GDB は最初にこれを探します。GDB がホームディレクトリ内を検索する場所はいくつかあり、これらの場所は順番に検索され、最初に見つかったファイルのみをロードします。MacOS 以外では以下の場所が検索されます。

- `$XDG_CONFIG_HOME/gdb/gdbealyinit`
- `$HOME/.config/gdb/gdbealyinit`
- `$HOME/.gdbealyinit`

`-nx, -n` オプションでこれらの初期初期化ファイルを読むことを阻止できます。

2.1.4.2 システム全体の初期化ファイル

以下の二か所が検索され、これらは常にチェックされます。

system.gdbinit

単一のシステム全体初期化ファイルです。 `--with-system-gdbinit` オプションで設定できます。

system.gdbinit.d

ディレクトリです。

2.1.4.3 ホームディレクトリ初期化ファイル

システム全体初期化ファイルを読んだ後、これを探します。以下の場所を検索し、最初に見つかったファイルのみをロードします。 MacOS 以外では以下の場所が検索されます。

- `$XDG_CONFIG_HOME/gdb/gdbinit`
- `$HOME/.config/gdb/gdbinit`
- `$HOME/.gdbinit`

2.1.4.4 ローカルディレクトリ初期化ファイル

カレントディレクトリで`.gdbinit` ファイルを検索します。`-x, -ex` で指定したコマンドを除いて最後にロードされます。すでにホームディレクトリ初期化ファイルとして読み込まれている場合は再度ロードされることはありません。

2.2 GDB の終了

GDB を終了するには `quit [expression]`, `exit [expression]` または `q` または `ctrl+d` で終了できます。`expression` に指定した値は終了コードとして帰ります。`ctrl+c` は実行中の GDB コマンドアクションを終了します。

2.3 シェルコマンド

GDB 起動中にシェルコマンドを使用することができます。

```
shell <command-string>
!<command-string>
```

`pipe` 命令を使用して gdb の出力を他のプログラムに繋ぐことができます。

```
pipe [command] | <shell_command>
| [command] | <shell_command>
pipe -d <delim> <command> <delim> <shell_command>
| -d <delim> <command> <delim> <shell_command>
```

`command` が `|` を含むときには -d で別の記号(列)を指定します。

2.4 ロギング出力

GDB の出力をファイルに行うことができます。GDB にはロギングを制御するコマンドがいくつか用意されています。

set loggging enabled [on|off] ロギングのオンオフ切り替え

set logging file <file> 現在のログファイルの名前を変更。デフォルト値は gdb.txt

set logging overwrite [on|off] 上書きか書き足しか(on で上書き)。デフォルト値は `off`

set logging redirect [on|off] on にすると GDB の出力がログファイルにのみ行われる。デフォルト値は `off`

set logging debugredirect [on|off] on にすると GDB デバッグの出力がログファイルにのみ行われる。デフォルト値は `off`

show logging ロギングの設定を表示する

3 GDB コマンド

GDB コマンドは曖昧性がなければコマンド名の最初の数文字のみで使用できます。また、`ret`(エンター)を入力すると特定の GDB コマンドを繰り返し実行できます。また、TAB キーによる補完機能が有効です。

3.1 コマンド構文

GDB コマンドは一行の長さ無制限の入力です。`command [args]` の形をしています。

`run` など一部コマンドを除いて空白行を入力すると直前のコマンドを繰り返します。`list` 及び `x` コマンドでは引数が変わります(???)。

3.2 コマンド設定

多くのコマンドは変数及び設定で動作が変わります。これらの設定は `set` コマンドで変更できます。

gdbinit ファイルに書き込むことで初期化時に設定できますし、対話中にコマンドを実行して設定することもできます。

`with` コマンドを使用して、コマンド呼び出しの期間中一時的に設定を変更することもできます。

```
with <setting> [value] [-- command]
w <setting> [value] [-- command]
```

3.3 コマンド補完

GDB では TAB キーによる補完が有効です。候補が唯一の場合は自動で入力が保管され、複数ある場合は候補が表示されます。TAB を二回押して候補を表示する代わりに `esc ?`で表示することもできます。

以下のコマンドで補完候補の最大数を設定できます。デフォルト値は 200 です。

```
set max-completions <limit>
```

`limit` には整数値または `unlimited` が指定できます。

```
show max-completions
```

で現在の設定を確認できます。

3.4 ファイル名引数

ファイル名をコマンドの引数として渡す場合、ファイル名に空白、ダブルクォート、シングルクォートが含まれていない場合は単純な文字列として記述できます。これらが含まれている場合、いくつか方法があります。

- GDB に任せる
- エスケープを使う
- クオートで囲う

3.5 コマンドオプション

一部コマンドは先頭に-がついたオプションを受け付けます。コマンド名と同様に、明確な場合は省略形を使うことができます。また、補完も効きます。

一部コマンドの引数にハイフンを含む場合は--を使うことでそれ以降の引数をオプションとして解釈しなくなります。

3.6 ヘルプ

help コマンドを使用してコマンドのヘルプを閲覧できます。

```
help, h
```

引数なしの help コマンドはコマンドのクラスのリストを表示します。

```
help <class>
```

ヘルプクラスを指定するとそのクラスの個々のコマンドのリストを表示します。

```
help <command>
```

コマンドを指定するとそのコマンドの短い使用方法を表示します。

```
apropos [-v] <regexp>
```

コマンド、エイリアス及びそのドキュメントを検索し、引数で指定した正規表現を検索します。見つかったすべてを表示します。-v オプションをつけるとドキュメントの一致部分をハイライトして表示します。

```
complete <args>
```

コマンドの先頭部分の一致候補を表示します。

info, show. set コマンドを使用して、プログラムの状態や GDB の状態を設定および照会することができます。

```
info, i
```

プログラムの状態を表示します。help info でサブコマンドの一覧を閲覧できます。

```
set
```

式の結果を環境変数に割り当てます。

```
show
```

GDB の状態を表示します。set できるものは大体 show できます。

show にあって set できないものを以下に示します。

- version

バージョン情報を表示します。

- copying

著作権表示を行います。

- warranty

保証情報を表示します。

- configuration

GDB のビルド情報を表示します。

4 GDB でプログラムを実行する

GDB でプログラムを実行するにはコンパイル時にデバッグ情報を付与する必要があります。

任意の環境で、引数を指定して GDB を起動できます。ネイティブデバッグではプログラムの IO をリダイレクトしたり、実行中プロセスをデバッグしたり、子プロセスを強制終了したりできます。

4.1 デバッグのためのコンパイル

プログラムを効果的にデバッグするにはコンパイル時にデバッグ情報を生成する必要があります。この情報はオブジェクトファイルに保存され、各変数、関数のデータ型と実行可能コード内のソース行番号とアドレスの対応関係が記述されます。

デバッグ情報の生成は`-g` オプションで行うことができます。

GCC では`-g` オプションは`-O`(最適化オプション)と併用できます。

4.2 プログラムの開始

`run, r`

GDB でプログラムを実行するには `run` コマンドを使用します。このコマンドを使用するには GDB 起動時またはコマンドでプログラムを指定する必要があります。

引数

`run` コマンドの引数はそのままプログラムのコマンドライン引数として渡されます。

環境

プログラムは GDB から環境を継承します。`set` コマンドで環境を変更することもできます。

作業ディレクトリ

`set cwd` でプログラムの作業ディレクトリを設定できます。設定しない場合、GDB の作業ディレクトリを引き継ぎます。リモートデバッグの場合にはリモートサーバの作業ディレクトリを引き継ぎます。

標準入出力

通常、プログラムの標準入出力は GDB と同じになります。`tty` コマンドで別のデバイスを設定することもできます。

`run` コマンドで実行したプログラムは直ちに実行を開始します。

GDB はシンボルファイルの変更を検出し、再読み込みを行います。

```
  start
```

`start` コマンドはメインプロシージャにブレークポイントを設置して `run` します。引数の扱いは `run` と同様です。

```
  starti
```

start と同様ですが、ブレークポイントの位置は最初の命令です。

```
set exec-wrapper <wrapper>
show exec-wrapper
unset exec-wrapper
```

を使用してデバッグ用プログラムの起動します。つまり Shell コマンド exec wrapper program を実行します。

```
set startip-with-shell
set startip-with-shell on
set startip-with-shell off
show startip-with-shell
```

プログラムをシェルで実行します。

```
set auto-connect-native-target
set auto-connect-native-target on
set auto-connect-native-target off
show auto-connect-native-target
```

現在の下位プロセスがターゲットに接続されていないときにローカルマシンで実行します。

```
set disable-randomization
set disable-randomization on
```

プログラムの仮想アドレス空間のネイティブランダム化をオフにします。

4.3 プログラムの引数

プログラムへの引数は run コマンド実行時に指定します。指定しなかった場合は以前の run 実行時の引数または set args で指定した引数を使用します。

```
set args
show args
```

4.4 プログラムの環境

環境は、環境変数とその値のことを指します。

```
path <directory>
```

環境変数の先頭にディレクトリを追加します。GDB が使用する環境変数の値は変化しません。

```
show paths
```

実行可能ファイルの検索パスのリストを表示します。

```
show environment [varname]
```

環境変数 varname の値を表示します。指定しない場合はすべての環境変数とその値を表示します。

```
set environment varname [=value]
```

環境変数 varname の値を設定します。値はプログラムに対してのみ変更され、GDB が読む変数の値は変わりません。

4.5 プログラムの作業ディレクトリ

```
set cwd [directory]
```

下位の作業ディレクトリを directory に変更します。

```
show cwd
```

下位プロセスの作業ディレクトリを表示します。

```
cd
```

GDB の作業ディレクトリを変更します。

```
pwd
```

GDB の作業ディレクトリを表示します。

4.6 プログラムの入出力

デフォルトでは GDB が実行するプログラムは GDB と同じターミナルに入出力を行います。

```
info terminal
```

プログラムが使用している端末モードについての情報を表示します。

run コマンドでシェルのリダイレクト機能を使えます。

```
run > <output file>
```

tty コマンドで入出力の場所を指定できます。

```
tty <file>

set inferior-tty <tty>
show inferior-tty
```

デバッグ対象のプログラムの tty を設定、表示します。

4.7 すでに実行中のプロセスのデバッグ

```
attach <pid>
```

すでに実行中のプロセスのプロセス ID を指定してデバッガをアタッチします。

```
set exec-file-mismatch 'ask|warn|off'
show exec-file-mismatch
```

GDB がロードした実行ファイルとアタッチしたプログラムの実行ファイルが一致するか確認した際に不一致だった場合の挙動を設定します。 ask は警告を出しプロセスの実行ファイルをロードするか確認します。warn は警告を表示のみします。off は不一致確認を行いません。

GDB はプロセスにアタッチするとそのプロセスを停止します。続行するには continue をします。

```
detach
```

プロセスからデタッチします。

4.8 子プロセスの終了

```
kill
```

GDB で実行されている子プロセスを終了します。

4.9 複数の下位接続とプログラムのデバッグ

GDB では 1 セッションで複数のプログラムを実行、デバッグできます。一部システムでは同時に行えます。

GDB では各プログラムの状態を inferior と呼ばれるオブジェクトで管理します。

```
info inferior
```

現在存在する inferior を表示します。

```
inferior
```

現在の inferior の情報を表示します。

```
info connection
```

現在開いているターゲット接続を表示します。

```
inferior <infno>
```

現在の inferior を infno に変更します。

```
add-inferior [-copies <n>] [-exec <executable>] [-no-connection]
```

実行ファイルを `executable` とする inferior を n 個追加します。

```
clone-inferior [-copies <n>] [infno]
```

inferior を n 個コピーします。

```
remove-inferiors infno...
```

inferior を削除します。

```
detach inferior infno...
kill inferior infno...
```

inferior を指定して detach, kill します。

```
set print inferior-events [on|off]
show print inferior-events
```

inferior のプロセスが開始または終了したときに通知を受け取ります。

```
maint info program-spaces
```

GDB によって管理されているすべてのプログラムスペースのリストを出力します。

4.9.1 inferior 固有のブレークポイント

複数の inferior プロセスをデバッグする場合、すべての inferior プロセスにブレークポイントを設定するか、個別にするか選択できます。

```
break <locspace> inferior inferior-id
break <locspace> inferior inferior-id if ...
```

4.10 複数スレッドのプログラムのデバッグ

GDB はマルチスレッドデバッグを行うために以下の機能を提供します。

- 新しいスレッドの自動通知
- スレッドを切り替えるコマンド
- 既存のスレッドの情報を表示するコマンド
- スレッドのリストにコマンドを適用するコマンド
- スレッド固有のブレークポイント
- スレッド開始、終了時のメッセージ設定

GDB では複数スレッドを観察できます。今見えているスレッドをカレントスレッドといいます。デバッグコマンドはカレントスレッドに適応されます。

新しいスレッドを検出するとスレッド識別子を表示します。

```
  info threads [-gid] [thread-id-list]
```

スレッドの情報を表示します。引数を指定しなければすべてのスレッドの情報が表示されます。-gid を指定するとグローバルスレッド番号も表示します。

```
  thread <tid>
```

カレントスレッドを tid のスレッドに変更します。

```
  thread apply [thread-id-list | all [-ascending]] [flag]... <command>
```

指定したスライド全体にコマンドを適用します。フラグに設定できるのは以下のとおりです。

-c コマンド内のエラーを表示してその後のコマンドは続行されます。
-s コマンド内のエラーを無視します。
-q スレッド情報を表示しません。

```
  taas [option]... <command>
```

thread apply all -s [option]... <command>のショートカット。

```
  tfaas [option]... <command>
```

thread apply all -s -- frame apply all -s [option]... <command>のショートカット。

```
  thread name [name]
```

カレントスレッドに名前をつけます。名前を指定しない場合は名前を削除します。

```
  thread find [regexp]
```

指定した正規表現と一致する名前またはシステムタグを持つスレッドを検索して表示します。

```
  set print thread-event [on|off]
  show print thread-event
```

スレッド開始、終了時のメッセージを有効または無効にします。

4.11 フォークのデバッグ

```
  set follow-fork-mode <mode>
```

fork の呼び出しに対する応答を設定します。

- parent 元のプロセスは fork のあとデバッグされます。子プロセスは妨げられずに実行されます。デフォルト。
- child 新しいプロセスが fork のあとにデバッグされます。親プロセスは妨げられずに実行されます。

```
  set detach-on-fork 'on|off'
```

- on 子プロセスは切り離され、独立して実行されます。(デフォルト)

- `off` 両方のプロセスが GDB の制御化に置かれます。片方のプロセスは中断されます。(別 inferior)

4.12 チェックポイント、再起動

デバッグ中の状態にチェックポイントをおいて戻ることができます。

```
checkpoint
```

現在の状態にチェックポイントを起きます。

```
info checkpoint
```

設置されたチェックポイントの一覧を表示します。

```
restart <checkpoint-id>
```

チェックポイントに戻ります。

```
delete checkpoint <checkpoint-id>
```

チェックポイントを削除します。

5 停止と継続

デバッガを使用することでプログラムを停止することができます。

5.1 ブレークポイント、ウォッチポイント、キャッチポイント

- ブレークポイント

プログラム中の特定の場所で、そこに到達すると停止する。

- ウォッチポイント

式の値が変化したときに停止する。

- キャッチポイント

例外やライブラリロードなどで停止する。

5.1.1 ブレークポイントの設定

ブレークポイントは `break, b` コマンドで設定できます。さらに変数 `$bpnum` にブレークポイントの数が保存されています。

一つのブレークポイントが複数のコードの位置にマッピングされることがあります。例えば C++の template やオーバーロードなど。その場合、設定時にその数を出力します。

デバッグ中のプログラムがブレークポイントに到達すると、変数 `$_hit_bpnum` と `$_hit_locno` がセットされます。

```
break, b
```

引数無しで `break` コマンドを実行すると、選択されたスタックフレームの次に実行される命令にブレークポイントが設置されます。

```
break ... [-force-condition] if <cond>
```

条件付きブレークポイントを設定します。このブレークポイントに到達したとき、`cond` の式がゼロでない場合にプログラムは停止します。 `-force-condition` を指定すると、式 `cond` が無効な式でもブレークポイントを設置します。

`tbreak args`
一回限りのブレークポイントを設置します。引数は `break` と同じです。このブレークポイントに一回プログラムが到達するとそのブレークポイントは自動的に消去されます。
`hbreak args`
ハードウェアブレークポイントを設置します。
`thbreak args`
一回限りのハードウェアブレークポイントを設置します。
`rbreak <regex>`
正規表現 `regex` にマッチするすべての関数にブレークポイントを設置します。`regex` に`.`を指定すればすべての関数にブレークポイントを設置できます。
`rbreak <file>:<reex>`
ファイル名を指定して `rbreak` を実行します。
`info breakpoints [list...]`
`info break [list...]`
全てのブレークポイント、ウォッチポイント、トレースポイント、キャッチポイントを表示します。 `list` を指定すると指定したものだけを表示できます。

ブレークポイントは共有ライブラリを読み込んだ場合などに再計算されます。また、共有ライブラリロード以前にブレークポイントを設定しておくことも可能です。
`set breakpoint pending auto`
通常の動作です。GDB がロケーションを解決できない場合、作成するかどうかをユーザに問い合わせます。
`set breakpoint pending on`
`set breakpoint pending off`
on の場合、解決できなくても作成します。off ではしません。 以上の設定はブレークポイントを設定するときにだけ適用されます。一度設置されたブレークポイントは自動で再計算されます。
`set breakpoint auto-hw 'on|off'`
自動でハードウェアブレークポイントを使用するかどうかの設定です。
`set breakpoint always-inserted 'off|on'`
off がデフォルト値です。プログラムが停止したときにブレークポイント用に書き換えたプログラムコードを元に戻すかどうかの設定です。

5.1.2 ウォッチポイントの設定
ウォッチポイントで監視できるものは以下の通りです。
- 単一の変数
- 適切なデータ型にキャストされたアドレス
- 式

ウォッチポイントはその計算が可能になる前から設定できます。そして有効な値になったときにプログラムを停止します。

5.1.3 キャッチポイントの設定

キャッチポイントを使用することでプログラムの例外や共有ライブラリロードなどのイベントによりデバッガに停止させることができます。

```
  catch <event>
```

event が発生すると停止します。イベントは以下の通りです。

```
  throw [regexp]
  rethrow [regexp]
  catch [regexp]
```

C++の例外が投げられた、再び投げられた、キャッチされた。 regexp が与えられている場合、その正規表現にマッチする例外だけがキャッチされます。

```
  syscall [name | number | group:groupname | g:groupname]
```

システムコール発行時または復帰。

```
  fork
  vfork
```

fork および vfork 呼び出し

```
  load [regexp]
  unload [regexp]
```

共有ライブラリの読み込み、アンロード

```
  signal [signal... | 'all']
```

シグナル発行。

```
  tcatch
```

一回限りのキャッチポイントを設置

5.1.4 ブレークポイントの削除

```
  clear [locspec]
```

引数を指定しない場合、次の命令のブレークポイントを削除します。指定した場合、そのブレークポイントを削除します。

```
  delete [breakpoints] [list...]
```

引数で指定したブレークポイント、ウォッチポイント、キャッチポイントを削除します。引数を指定しない場合、すべて削除します。

5.1.5 ブレークポイントを無効にする

ブレークポイント、ウォッチポイント、キャッチポイントには以下の状態があります。

- 有効
  - 有効なブレークポイント。
- 無効
  - 無効なブレークポイント。
- 一度だけ有効
  - 一度プログラムが停止すると無効になる。
- 何回か有向
  - 指定した回数プログラムが停止すると無効化される。

- 有効のち削除
  - 一度プログラムが停止すると削除される。

```
disable [breakpoints] [list...]
enable [breakpoints] [list...]
```

指定したブレークポイントを無効化、有効化する。 disable は引数を指定しない場合、何もおこらない。enable は全て有効になる。

```
enable [breakpoints] once <list...>
enable [breakpoints] count <count> <list...>
enable [breakpoints] delete <list...>
```

一度だけ有効、何回か有効、有効のち削除にする。

5.1.6 ブレーク条件

ブレークポイントには条件を付けてそれを満たす場合のみプログラムを停止することができます。

```
condition [-force] <bnum> <expression>
```

bnum のブレークポイントに条件を付与します。`-force` オプションをつけると現時点で無効な式も使用できます。 `expression` を指定せずに実行すれば条件式を外すことができます。

```
ignore <bnum> <count>
```

ブレークポイントに到達した `count` 回は無視して、次からは停止します。

5.1.7 ブレークポイントコマンドリスト

ブレークポイントで停止したときに実行するコマンドを指定することができます。

```
commands [list...]
... <command-list> ...
end
```

`list` で指定したブレークポイントにコマンドリストを割り付けます。削除するにはコマンドリストを指定せずに実行します。

5.1.8 動的 printf

動的 printf`dprintf` ブレークポイントと printf を組み合わせたようなコマンドです。

```
dprintf <locspec>, <template>, <expression> [, <expression>...]
```

locspec で指定した場所にプログラムが到達すると `template` に従って式の `expression` を出力します。

```
set dprintf-style <style>
```

dprintf の以下のスタイルを指定します。

- `gdb`

GDB の printf のハンドル。`%V` の指定子が使える。

- `call`

ユーザプログラムの関数を使用します。通常は `printf`。`%V` は使えません。

- `agent`

リモートデバッグエージェントに出力させます。%V は使えません。

```
  set dprintf-function <function>
```

call のときに使用する関数を設定します。

```
  set dprintf-channel <channel>
```

channel に空でない値を設定すると、fprintf-function の第一引数にそれを与えて評価します。

```
  set disconnected-dprintf 'on|off'
```

agent のときに、ターゲットが切断されたときに dprintf の実行を続けるかどうかの設定です。

5.1.9 ブレークポイントをファイルに保存する方法

```
  save breakpoints [<filename>]
```

filename のファイルにブレークポイントを保存します。この際、コマンドやカウンタも保存されます。 これを読むには、source コマンドを使用します。式付きのウォッチポイントは無効で失敗する場合があります。

5.1.10 静的プローブポイントの一覧表示

GDB は SDT(Statically Defined Tracing; 静的定義トレース)をサポートしています。プローブは小規模なランタイムコードやフットポイント、動的再配置をデザインします。

現在以下のタイプのプローブが ELF 互換システムで実装されています。

- SystemTap アセンブリ、C、C++に対応
- DTrace C、C++に対応

5.1.11 ブレークポイントを挿入できません。

ハードウェアブレークポイントを挿入しすぎるとこのエラーが出ます。無効化または削除してください。

5.2 継続とステップ

継続は通常の実行と同様にプログラムが終了するまで実行することで、ステップは 1″ステップ"だけ実行することを意味します。ここで 1 ステップは一行だったり位置命令だったりします。

```
  continue [ignore-count]
  c [ignore-count]
  fg [ignore-count]
```

次の停止場所まで継続実行します。ignore-count を指定するとその回数分ブレークポイントを無視します。3 つのコマンドは完全に同じ動作をします。

```
  step [count]
  s [count]
```

ソースコード上の次の行までを実行します。関数呼び出しでは、その内部にデバッグ情報があればそこで停止します。また、停止する位置はソース行の最初の命令です。

step は関数の行番号情報がある場合にだけ関数に入ります。

count を指定するとその回数分 step を実行します。

```
  next [count]
```
現在のスタックフレーム内の次のソースコード行に進みます。つまり関数の内部では停止しません。
```
  set step-mode [on|off]
```
on のときデバッグ行情報を含まない関数の最初の命令で停止します。off の場合にはデバッグ情報を含まない関数はスキップされます。off がデフォルトです。
```
  finish
  fin
```
選択したスタックフレームの関数がリターンするまで実行を続けます。返り値がある場合にはそれを表示します。
```
  set print finidh [on|off]
```
finish 終了時に返り値を表示するかどうかの設定です。デフォルトは on です。
```
  until
  u
```
現在のスタックフレームで、現在の行を過ぎたソース行に到達するまで継続実行する。until 中にジャンプ命令があったとき、PC がジャンプアドレスより大きくなるまで継続することを除いて next と同じである。ループの最後の行で until をすればループを抜けるまで実行することができるが、機械語の配置によっては直感通りの動作をしない場合がある。
```
  until <locspec>
  u <locspec>
```
引数で指定した位置に到達するまたは現在のスタックフレームが戻るまで継続実行する。あくまで現在のスタックフレームで停止する。
```
  advance <locspec>
```
引数で指定した位置まで継続実行する。until とことなり、スタックフレームは無関係である。
```
  stepi [arg]
  si
```
機械語命令を一つ実行します。引数は step とおなじです。
```
  nexti [arg]
  ni
```
機械語命令を一つ実行します。call 命令の場合は帰るまで実行します。引数は next と同じです。

5.3 関数とファイルのスキップ

```
  skip [options]
```
指定されたものをスキップします。指定できるものは以下です。

- `-file <file>, -fi <file>`: file 内にある関数すべてをスキップします。

- gfile <file-glob-pattern>, -gfi <file-glob-pattern>: パターンにマッチするファイル内の関数をスキップします。
- function <linespec> -fu <linespec>: 指定場所を含むまたは名前を持つ関数をスキップします。
- rfunction <regexp>, -rfu <regexp>: 正規表現 regexp にマッチする関数をスキップします。
- 指定なし: 現在の関数をスキップします。

```
skip function [linespec]
skip file [file]
```

このコマンドの実行後、ステップ実行では指定した関数またはファイル内の関数がスキップされます。ファイル名、linspec を指定しない場合、現在の関数/ファイルが指定されます。

```
info skip [range]
```

指定されているスキップ対象を表示します。

```
skip delete [range]
skip enable [range]
skip disable [range]
```

スキップ対象を削除、有効化、無効化します。

```
set debug skip [on | off]
```

ファイルや関数のスキップに関するデバッグ出力をするかどうかの設定です。

5.4 シグナル

GDB にはプログラム中で発生したシグナルを検出する機能があります。通常ではエラーでないシグナルはそのままプログラムに渡されます。

```
info signals <sig>
info handle
```

すべてのシグナルと、GDB がどう扱うかの表を出力します。<sig>を指定した場合はそれのみを出力します。

```
catch signal [signal...|'all']
```

シグナルにキャッチポイントを設定します。

```
handle signal [signal...] [keywords...]
```

シグナル検出時の挙動を設定します。キーワードに設定できるものは以下です。

- nostop: そのままプログラムの実行を継続します。print と併用できます。
- stop: プログラムを停止します。print と併用できます。
- print: シグナル検出の旨を表示します。
- noprint: シグナル検出の旨を表示しません。
- pass, noignore: シグナルをプログラムに渡します。
- nopass, ignore: シグナルをプログラムに渡しません。

5.5 スレッドストップ

5.5.1 オールストップモード

このモードでは GDB がプログラムを停止するたびに他のすべてのスレッドも停止する。再開する際も同様にすべてのスレッドが再開する。

```
set scheduler-locking <mode>
```

スケジューラロッキングモードを設定します。

- `off`: すべてのスレッドはいつでも実行できる。
- `on`: 実行再開時には現在のスレッドだけが再開する。
- `step`: ステップ実行時には `on`、それ以外では `off` のような動作をする。`continue, until, finish` などで復帰できる。
- `replay`: replay モードでは `on` それ以外では `off` のような動作をする。

```
set schedule-multiple
```

実行コマンド発行時に、複数プロセスのスレッドの再開を許可するモードの設定。

5.5.2 ノンストップモード

このモードでは GDB がプログラムを停止すると、その当該スレッドのみが停止します。さらに実行コマンド各種は現在のスレッドにのみてきようされます。

ノンストップモードに入るには以下のコマンドを使用します。

```
set pagination off
set non-stop on
```

```
set non-stop 'on|off'
```

ノンストップモードをオン、オフにします。

```
continue -a
```

ノンストップモードで、すべてのスレッドに `continue` を適用します。`continue` 以外の実行コマンドは現在、`-a` をサポートしていません。

5.5.3 プログラムを非同期実行する

GDB の実行コマンドにはフォアグラウンド動作とバックグラウンド動作があります。フォアグラウンドではプログラムがあるスレッドが停止したことを報告するのを待ってから別のコマンドのプロンプトを表示します。バックグラウンドでは直ちにコマンドプロンプトを表示します。これにより実行中にコマンドを発行することができます。

ターゲットが非同期モードをサポートしていない場合、バックグラウンド系のコマンドはエラーを吐きます。

バックグラウンド実行を指定するには、コマンドに `&` をつけます。例えば `continue&` など。 バックグラウンドを受け付けるコマンドは以下のとおりです。

- `run`
- `attach`
- `step`

- stepi
- next
- nexti
- continue
- finish
- until

バックグラウンド実行中のプログラムを中断するには以下のコマンドを使用します。

```
interrupt [-a]
```

オールストップモードではすべての、ノンストップモードでは現在のスレッドを一時停止します。 -a をつけるとノンストップモードでもすべてのスレッドでプログラムを停止します。

5.5.4 スレッド固有のブレークポイント

プログラムが複数のスレ度を持つ場合、すべてのスレッドまたは特定のスレッドにブレークポイントを設置することができます。

```
break <locspec> [thread <thread-id>] [if ...]
```

thread <thread-id>を指定しない場合、すべてのスレッドにブレークポイントを設置します。指定した場合はそのスレッドにのみ設置されます。

スレッド固有のブレークポイントはそのスレッドがなくなった場合、自動的に削除されます。

5.5.5 システムコール割り込み

マルチスレッドプロセスのデバッグに GDB を使用すると副作用により、ブレークポイント停止やその他イベントのために使用するシグナルが原因でシステムコールが早期リターンする場合があります。

この問題はプログラム側でシステムコールの返り値を使用して対応する必要があります。

5.5.6 オブザーバモード

ノンストップモードでビルドし、GDB による中断の影響を排除するためには変数を設定して、メモリ書き込みやブレークポイントの挿入などデバッガが状態を変更する動作をすべて無効にすることができます。

```
set observer 'on|off'
```

on の場合、以下のすべての変数が無効になり、ノンストップモードになります。off にすると通常のデバッグに復帰しますが、ノンストップモードのままです。

```
set may-write-register 'on|off'
```

GDB が print の代入式やジャンプコマンドでレジスタの値を変更するかどうかの制御。デフォルトは on です。

```
set may-write-memory 'on|off'
```

GDB が print の代入式などでメモリの内容を変更するかどうかの制御。デフォルトは on です。

```
set may-insert-breakpoints 'on|off'
```

GDB がブレークポイントの挿入を試みるかどうかの制御。デフォルトは on です。
  set may-insert-tracepoints 'on|off'
GDB がトレースポイントの挿入を試みるかどうかの制御。デフォルトは on です。
  set may-insert-fast-tracepoints 'on|off'
GDB がファストトレースポイントの挿入を試みるかどうかの制御。デフォルトは on です。
  set may-interrupt 'on|off'
GDB が割り込みを試みるかどうかの制御。デフォルトは on です。

6 後ろ向きのプログラム実行

サポートされている環境であれば、プログラムの実行を元に戻すことができます。
  reverse-continue [ignore-count]
  rc [ignore-count]
逆順に continute します。
  reverse-step [count]
逆順に step します。
  reverse-stepi [count]
逆順に stepi します。
  reverse-next [count]
逆順に next します。
  reverse-nexti
逆順に nexti します。
  reverse-finish
現在の関数の呼び出しポイントまで戻ります。
  set exec-direction 'reverse|forward'
ターゲットの実行向きを指定します。

7 Inferior の実行の記録と再生

いくつかの OS ではレコードすることによって逆順の実行ができます。
 record <method>
メソッドを指定してレコードします。メソッドは以下のとおりです。

- full: GDB ソフトウェアによる完全なレコードです。
- trace <format>: ハードウェア命令によるレコードです。Intel プロセッサでサポートされます。データはリングバッファに書き込まれるので限定的な巻き戻しのみ可能です。フォーマットは以下のとおりです。
  - bts: Branch Trace Store。
  - pt: Intel Processor Trace。実行トレースを圧縮して保存します。

record コマンドを使うにはプログラムを実行しておく必要があります。

ノンストップモードまたは非同期実行モードでは full はサポートされません。

 record stop
レコードを停止します。ログはすべて削除され、Inferior は終了するか最終状態のま

まになります。リプレイモード中にこれを発行するとその時点から通常のデバッグに復帰します。

`record goto 'begin|start|end'|<n>`

指定した場所に戻ります。begin と start は同じ場所です。n は n 番目の命令です。

`record save <filename>`

レコードを保存します。

`record restore <filename>`

ファイル名から実行ログをリストアします。`save` で保存したものを読みます。

`set record full insn-number-max <limit>|'unlimited'`

最大のレコード容量を設定できます。デフォルトでは 200000 です。レコードが最大容量に達すると、一番最初の命令から順番に削除しながらレコードが進みます。`limit` に 0 または `unlimited` が設定された場合、命令は削除されません。

`set record full stop-at-limit 'on|off'`

レコードが最大容量に達したときに停止して続行するかどうかを尋ねます。

`set record full memory-query 'on|off'`

GDB が `full` のレコードをするとき、命令によって引き起こされたメモリ変更を記録できない場合の動作を制御します。on の場合にはどうするかを尋ね、off の場合には無視します。

`set record btrace replay-memory-access 'read-only|read-write'`

リプレイ中にメモリにアクセスする際の btrace レコードメソッドの動作を制御します。`read-only` の場合、GDB は readonly メモリへのアクセスのみを許可します。`read-write` の場合、GDB は readonly および readwrite メモリへのアクセスを許可します。

`set record btrace cpu <identifier>`

プロセッサ・エラッタを回避するために使用するプロセッサを設定します。プロセッサ・エラッタとは、設計や製造に起因するプロセッサ動作の血管のことを指します。

引数の `identifier` は CPU 識別子で `vendor:processor identifier` という形か `none`, `auto` が指定できます。

`set record btrace bts buffer-size <size>|'unlimited'`

BTS 形式でのブランチトレースに要求されるリングバッファのサイズを指定します。デフォルト値は 64KB です。

`set record btrace pt buffer-size <size>|'unlimited'`

IPT でのリングバッファのサイズを設定します。デフォルト値は 16KB です。

`info record`

レコード方式によってさまざまな統計情報を表示します。

`record delete`

レコード大賞が過去で実行された場合、それ以降のログを削除し、現在のアドレスからレコードを再開します。

`record instruction-history`

レコードされたログから命令を逆アセンブルします。

`set record instruction-history-size <size>|'unlimited'`

`record instruction-history` で表示される命令の数を設定します。

`record function-call-history`
関数単位で実行履歴を表示します。
`set record function-call-history-size <size>|'unlimited'`
`record function-call-history` で表示される数を設定します。

8 スタックの検証

8.1 スタックフレーム

コールスタックはスタックフレームに分割され、各フレームは関数呼び出しに関するデータを保持します。

GDB は既存のスタックフレームにそれぞれレベルをつけます。レベルは最も内側のフレームが 0、それを呼んだフレームが 1…という風に付きます。

8.2 バックトレース

プログラムの呼び出し履歴をバックトレースと云います。

```
backtrace [option]... [qualifier]... [count]
bt [option]... [qualifier]... [count]
```

すべてのスタックフレームのバックトレースを表示します。`count` は正の値を指定すると内側いくつか、負の値では外側いくつかを表示します。 option に指定できるものは以下のとおりです。

- `-full`: ローカル変数の値も表示する。
- `-no-filters`: Python フレームフィルタを実行しません。
- `-hide`: Python フレームフィルタで elide にされたフレームを表示しません。
- `-past-main [on|off]`: main 以降もバックトレースを続けるかどうか。`backtrace past-main` で設定可。
- `-past-entry [on|off]`: エントリポイント移行もバックトレースを続けるかどうか。`backtrace past-entry` で設定可。
- `-entry-values 'no|only|preferred|if-needed|both|compact|default'`: 関数入力時の print の設定。`print entry-values` で設定可。
- `-frame-arguments all|scalars|none`: 非スカラーフレーム引数の print の設定。`print frame-arguments` で設定可。
- `-raw-frame-arguments [on|off]`: フレーム引数を生で表示するかどうか。`print raw-frame-arguments` で設定可。
- `-frame-info auto|source-line|location|source-and-location|location-and-address|short-location`: フレーム情報の print 設定。`print framw-info` で設定可。

qualifier は下位互換のための引数です。

マルチスレッド環境では、現在のスレッドのバックトレースが表示されます。複数のスレッドのバックトレースを表示するには `thread apply` を使用できます。

8.3 フレームの選択

```
frame [frame-selection-spec]
```

```
f [frame-selection-spec]
```
指定したフレームを選択します。指定子に指定できるものは以下です。

- `<num>, level <num>`: スタックフレームレベル。
- `address <stack-address>`: スタックアドレス。`info frame` で確認できます。
- `function <function-name>`: 関数名でスタックを指定します。
- `view <stack-address> [pc-addr]`: GDB のバックトレースの一部ではないフレームを表示する。

```
up [n]
down [n]
```
現在選択中のフレームの `n` 個上(外側)、下(内側)のフレームを選択します。

8.4 フレーム情報

```
info frame [frame-selection-spec]
info f [frame-selection-spec]
```
フレーム情報を表示します。
```
info args [-q] [-t <type_regexp>] [regexp]
```
選択されたフレームの引数を表示します。`-q` を指定するとヘッダー情報や引数が出力されなかった理由を説明するメッセージが非表示になります。 後ろ 2 つのオプションは引数の型または名前を指定できます。
```
info locals [-q] [-t <type_regexp>] [regexp]
```
選択されたフレームのローカル変数を表示します。オプションは `info args` と同じです。

8.5 それぞれのフレームにコマンドを適用する

```
frame apply [all|count|-count|level <level>...] [option]... <command>
```
指定したフレームにコマンドを適用します。 option に指定できるものは以下のとおりです。

- `-past-main`: main より先もバックトレースを続けます。
- `-past-entry`: エントリポイント以降もバックトレースを続けます。
- `-c`: エラーがあったときに表示して、継続します。
- `-s`: エラーがあったときに表示せずに、継続します。
- `-q`: フレーム情報を表示しません。

```
faas <comamnd>
```
`frame applu all -s <command>`のエイリアス。

9 ソースファイルの検証

9.1 リスト

```
list <linenum>
```

現在のソースコード行を中心に linenum 行のソースコードを表示します。

```
list <function>
```

関数の開始点を中心にコードを表示します。

```
list [+]
```

最後に出力された行の続きを表示する。

9.2 位置指定

9.3 編集

9.4 検索

9.5 ソースパス

9.6 機械語

9.7 ソース読み込み無効化