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。

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。

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。

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。

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

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。

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。

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。

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。

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。

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。

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

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。

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。

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。

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。

スレッドストップ

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。

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。

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。

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。

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。

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。