第 10 回 Unix ゼミ
C プログラム(デバッグ編)

川島研 B4 高木 空

2024 年 07 月 13 日

デバッガ

- デバッグ (debug)
  - バグ(bug)を取り除く(de-)こと
- デバッグの手法
  - print デバッグ
  - コードを読む
  - デバッガを使う
    - 今回の主題

デバッグの手法

- print デバッグ
  - ソースコードに print を埋め込む
  - 利点
    - 気軽に実行できる
    - 欲しい出力を欲しい形式で得られる
  - 欠点
    - ソースコードを改変する必要がある
    - バグの箇所を検討してからしかできない
    - 得られる情報が少ない

デバッグの手法

- デバッガを使う
  - デバッガ - デバッグを補助するツール
  - 利点
    - プログラム全体を観察できる
    - プログラムの変更が(一般には)不要
    - スタックやメモリの監視もできる
  - 欠点
    - 使い方を知っている必要がある

C 言語のデバッガ

- C 言語プログラムのデバッガ
- GDB
  - Gnu Project のデバッガ
  - gcc を使うならコレ
  - Linux に標準搭載されている
- LLDB
  - LLVM のデバッガ
  - clang を使うならコレ

GDB の起動

```
$ gdb [options] [<program>]
```

で GDB を起動

- `options`：起動時のオプションを指定
  - `--help`：簡単な使い方を表示
  - `--tui`：TUI モード(後述)で起動
- `program`：デバッグ対象の実行可能ファイルを指定

GDB の終了

- GDB が起動すると先頭に(gdb)と表示される

```
(gdb) quit [<expr>]
(gdb) exit [<expr>]
```

で GDB を終了(ctrl-d でも可)

引数:

- expr : GDB の終了コードを指定

GDB 起動中のシェルコマンド

```
(gdb) shell <command>
(gdb) ! <command>
```

で GDB 起動中にシェルコマンドを実行

引数:

- `command` : 実行するシェルコマンド

補足:

- パイプ等も使える

コマンド概要

- GDB はコマンドで操作
  - `quit` や `shell` もコマンド

```
(gdb) <command> [<args>...]
```

の形で入力

- コマンドが区別できれば省略できる
  - 例：`quit` → `q`
- `TAB` キーによる補完が可能
  - 候補が唯一の場合自動入力
  - 複数の場合 2 回押すと候補を表示

コマンド補助

```
(gdb) help [<class>|<command>]
```

コマンドの一覧や使い方を表示

引数:

- `class` : コマンド群を指定するクラス
- `command` : ヘルプを見たいコマンドを指定

補足:

- 引数無しで `help` を実行すると `class` の一覧が表示される

プログラムの開始

```
(gdb) run [<args>...]
```

でプログラムを GDB の下で実行

- `args` : プログラムのコマンドライン引数として渡される

チェックポイントとリスタート

特定の場所でのプログラムの状態を保存して再開できる

```
(gdb) checkpoint
```

で現在の状態を保存

```
(gdb) info checkpoints
```

で保存したチェックポイントの一覧を表示

```
(gdb) restart <id>
```

で指定したチェックポイントから再開

プログラムの停止

- GDB を使うとプログラムを中断できる
- 停止する条件
  - ブレークポイント
  - ウォッチポイント
  - キャッチポイント
- 実行の再開
  - 継続実行
  - ステップ実行

ブレークポイント

- プログラム上の指定場所に到達したら中断

```
(gdb) break [<loc>] [if <cond>]
```

でブレークポイントを設置

引数:

- `loc`：位置指定。以下の形式で指定:
  - `[<filename>:]<linenum>`：行番号指定
  - `<offset>`：行オフセット指定
  - `[<filename>:]<function>`：関数名指定
- `cond`：条件式。満たすときだけ中断

ウォッチポイント

式の値が変更したら中断

```
(gdb) watch [-location] <expr>
```

でウォッチポイントを設置

引数:

- `-location`：`expr` の参照するメモリを監視
- `expr`：監視対象の式

ブレークポイントの削除

```
(gdb) clear [<locspec>]
```

<locspec>にあるブレークポイントを削除

```
(gdb) delete [breakpoints] [<list>...]
```

<list>で指定したブレークポイント、ウォッチポイントを削除

```
(gdb) info breakpoints
```

設置されたブレークポイント、ウォッチポイントを表示

継続実行

次の停止場所まで実行する

```
(gdb) continue [<count>]
(gdb) fg [<count>]
```

で継続実行

引数:

- count : 停止箇所を無視する回数

ステップ実行

次の停止箇所を指定しつつ再開

```
(gdb) step [<count>]
(gdb) nexti [<count>]
```

で次の行まで実行。

補足:

- `step` は関数呼び出しの場合中に入る
- `next` は関数呼び出しの場合中に入らない

引数:

- `count` : 無視する行数

```
(gdb) until <locspec>
```

`locspec` で指定した位置まで実行

バックトレース

関数呼び出しのトレース

```
(gdb) backtrace
(gdb) where
(gdb) info stack
```

でバックトレースを表示

フレームの選択

```
(gdb) frame [<spec>]
```

でフレームを選択

引数:

- spec: フレームを指定。以下の形式が可能
  - <num>: フレーム番号を指定
  - <function-name>: 関数名を指定

```
up <n>
down <n>
```

で一つ上または下のフレームを指定

ステップ実行

```
(gdb) finish
```

で選択中のフレームが返るまで実行

ソースコード情報の表示

```
(gdb) list [<line>|<function>|+|-]
```

でソースコードを表示

引数:

- `line` : 行番号を指定してそこを中心に表示
- `function` : 関数名を指定して開始地点を中心に表示
- `+`, `-` : 前に表示した部分の後/前を表示

```
(gdb) list <start>, <end>
```

で指定部分を表示

プリント

```
(gdb) print [[<options>...] --] [/<fmt>] <expr>
```

でフォーマットを指定して expr の値を表示

引数:

- options : オプション
- fmt : フォーマット指定。以下が指定可能:
  - x, d, u, o, t, z: 16,10,符号なし 10,8,2,0 埋め 16 進数で表示
  - a: アドレスとして表示
  - c: 整数にキャストして文字として表示
  - f: 浮動小数として表示
  - s: 文字列として表示
  - r: 生フォーマットで表示
- expr : 表示する値

メモリ

```
(gdb) x[/<num><fmt><unit>] <addr>
```

でメモリの内容を表示

引数:

- num: 表示するメモリ量(単位: unit)
- fmt: フォーマット指定。以下が指定可能:
  - print で指定可能なフォーマット
  - i: 機械語命令として表示
  - m: メモリタグとして表示
- unit: num で使用する単位
  - b, h, w, g: 1, 2, 4, 8 バイト
- addr: 表示するメモリ領域の先頭アドレス

ディスプレイ

```
(gdb) display[/<fmt>] <expr>
```

でプログラムが停止する度に自動で表示

フォーマットに応じて `print` か `x` が呼ばれる

引数:

- `fmt`: フォーマットを指定。`print`,`x` で指定可能なものが指定可能
- `expr`: 表示する式またはアドレス

```
(gdb) info display
```

で設定されているディスプレイのリストを表示

```
(gdb) undisplay <dnum>...
```

でディスプレイを解除

人工配列

```
(gdb) p <first>@<len>
```

で first を最初の要素とする長さ len の配列として表示

例:

```
int *arr = (int*)malloc(2 * sizeof(int));
```

と宣言したものを

```
(gdb) p *arr@2
```

で表示

```
(gdb) p (int[2])*arr
```

でも可

レジスタ

```
(gdb) info registers
```

でベクタ、フロート以外のレジスタを全て表示

```
(gdb) info all-registers
```

ですべてのレジスタを表示

演習 1

演習 1 を解いてください。

LLDB の起動

```
$ lldb [<options>]
```

で LLDB を起動

LLDB の終了

- GDB が起動すると先頭に(lldb)と表示される

```
(lldb) quit [<expr>]
(lldb) exit [<expr>]
```

で LLDB を終了(ctrl-d でも可)

引数:

- expr : LLDB の終了コードを指定

コマンド概要

- LLDB はコマンドで操作
  - `quit` や `shell` もコマンド

```
(lldb) <noun> <verb> [-<option> [<option-value>]] [<args>]
```

の形で入力

- コマンドが区別できれば省略できる
  - 例：`quit` → `q`
- TAB キーによる補完が可能
  - 候補が唯一の場合自動入力
  - 複数の場合 2 回押すと候補を表示

コマンド補助

```
(lldb) help <command>
```

コマンドの一覧や使い方を表示

引数:

- `command` : ヘルプを見たいコマンドを指定

補足:

- 引数無しで `help` を実行すると `command` の一覧が表示される

プログラムの開始

```
(gdb) process launch [<options>] [<args>]
```

でプログラムを LLDB の下で実行

- `args` : プログラムのコマンドライン引数として渡される

`options`:

- `-s`: エントリポイントで停止

プログラムの停止

- LLDB を使うとプログラムを中断できる
- 停止する条件
  - ブレークポイント
  - ウォッチポイント
- 実行の再開
  - 継続実行
  - ステップ実行

ブレークポイント

- プログラム上の指定場所に到達したら中断

```
(gdb) breakpoint set [<options>]
```

でブレークポイントを設置

options:

- `-l <num>`: 行番号を指定
- `-n <name>`: 関数名を指定
- `-E <lang>`: 例外を指定

ウォッチポイント

式の値が変更したら中断

```
(gdb) watchpoint set expression [<options>] <expr>
(gdb) watchpoint set variable [<options>] <varname>
```

でウォッチポイントを設置

options:

- -w: ウォッチタイプを指定
  - read: 読まれたら停止
  - write: 書かれたら停止
  - read_write: 読み書きがあったら停止

ブレークポイントの削除

```
  (gdb) breakpoint delete [<options>] [<breakpoint-id-
list>]
  (gdb) watchpoint delete [<options>] [<breakpoint-id-
list>]
```

で指定したブレークポイント、ウォッチポイントを削除

options:

- -d: 現在無効なリストで指定した以外の全てを削除
- -f: 警告なしで全て削除

継続実行

次の停止場所まで実行する

```
(gdb) thread continue [<thread-index>]
```

で継続実行

ステップ実行

次の停止箇所を指定しつつ再開

```
(gdb) thread step-in
(gdb) thread step-over
```

で次の行まで実行。

補足:

- `step-in` は関数呼び出しの場合中に入る
- `step-over` は関数呼び出しの場合中に入らない

`options`:

- `-c <count>`: ステップ回数

バックトレース

関数呼び出しのトレース

```
(gdb) thread backtrace <options>
```

でバックトレースを表示

options:

- `-c <count>`: 表示するフレーム数
- `-s <index>`: 表示を開始するフレーム

フレームの選択

```
(gdb) frame select [<options>] [<frame-index>]
```

でフレームを選択

options:

- `-r <offset>`: 現在のフレームからのオフセットで指定

ステップ実行

```
(gdb) thread step-out
```

で選択中のフレームが返るまで実行

ソースコード情報の表示

```
(gdb) source list <options>
```

でソースコードを表示

options:

- `-l <linenum>`: 指定した行番号付近を表示
- `-f <filename>`: 指定したファイルを表示
- `-n <symbol>`: 指定した関数を表示

プリント

```
(gdb) frame variable [<options>] [<varname>...]
```

で選択中のフレームの局所変数の値を表示

options:

- -g: グローバル変数も表示
- -l: 局所変数を非表示
- -Z <len>: 配列として表示

レジスタ

```
(gdb) register read [<options>] [<register-name>]
```

でベクタ、フロート以外のレジスタを全て表示

options:

- -a: ベクタ、フロート含む全てのレジスタを表示

演習 2

演習 2 を解いてください。

プロファイラとは

- プロファイラ
  - プログラムの動作を記録し、動作の統計情報を調べるツール
- 使いどころ
  - 作成したプログラムの性能評価
  - ホットスポットの調査
  - ハードウェア性能情報の監視
- Perf
  - Linux 向けのプロファイラ

コマンド

```
# perf <command>
```

の形式でコマンドを実行

```
# perf
```

で command の一覧を閲覧

```
# perf help <command>
```

で各コマンドの使い方を表示

```
# perf list
```

でイベント(観測できる統計情報)の一覧を表示

stat

```
# perf stat [<options>] [<command>]
```

で command を実行して統計情報を表示

よく使う options:

- -B, --big-num：大きな数字を見やすく表示
- -e, --event <e>：集計するイベントを指定
  - カンマで区切って複数指定可

record

```
# perf record [<options>] [<command>]
```

で command を実行してプロファイル情報を収集

よく使う options:

- -e <events> : 収集するイベントを指定
- -o <filename> : 出力ファイル名を指定
- -g : コールグラフを有効化

report

```
# perf report [<options>]
```

で record で生成したプロファイル結果を調査

よく使う options:

- -i : 調査するファイルを指定
- --stdio : TUI モードを使用しない

演習 3

演習 3 を解いてください。