Proste i użyteczne kawałki gdb

Bartosz Szreder

Przechlapane

gdb to jeden z tych programów, którym przydałby się manpage do spisu treści jego manpage'a.

Przechlapane

gdb to jeden z tych programów, którym przydałby się manpage do spisu treści jego manpage'a.

. . . no może nie do końca

Flagi kompilacji

Kompilujemy program z `-O0 -ggdb`, chyba, że chcemy co chwila czytać *value optimized out*.

. . . no może nie do końca

Flagi kompilacji

Kompilujemy program z `-O0 -ggdb`, chyba, że chcemy co chwila czytać *value optimized out*.

- Uruchamianie debuggera: `gdb ./program`. To jeszcze nie uruchamia śledzonego programu, ale da konsolę debuggera.

. . . no może nie do końca

Flagi kompilacji

Kompilujemy program z `-O0 -ggdb`, chyba, że chcemy co chwila czytać *value optimized out*.

- Uruchamianie debuggera: `gdb ./program`. To jeszcze nie uruchamia śledzonego programu, ale da konsolę debuggera.
- Ustawianie argumentów dla śledzonego programu: `set args ...`. Coś takiego nie zadziała: `gdb ./program < argumenty`

. . . no może nie do końca

Flagi kompilacji

Kompilujemy program z `-O0 -ggdb`, chyba, że chcemy co chwila czytać *value optimized out*.

- Uruchamianie debuggera: `gdb ./program`. To jeszcze nie uruchamia śledzonego programu, ale da konsolę debuggera.
- Ustawianie argumentów dla śledzonego programu: `set args ...`. Coś takiego nie zadziała: `gdb ./program < argumenty`
- Uruchomienie programu: `run`. Można też tutaj podać argumenty do programu.

Ogólnie

- `set` ustawia różne parametry debuggera, `show` je wyświetla.

Ogólnie

- `set` ustawia różne parametry debuggera, `show` je wyświetla.
- `info` wyświetla informacje o śledzonym programie i różnych rzeczach, które podczas tego śledzenia ustawiamy (np. breakpointy).

Ogólnie

- `set` ustawia różne parametry debuggera, `show` je wyświetla.
- `info` wyświetla informacje o śledzonym programie i różnych rzeczach, które podczas tego śledzenia ustawiamy (np. breakpointy).
- `help <polecenie>` pomaga.

Ogólnie

- `set` ustawia różne parametry debuggera, `show` je wyświetla.
- `info` wyświetla informacje o śledzonym programie i różnych rzeczach, które podczas tego śledzenia ustawiamy (np. breakpointy).
- `help <polecenie>` pomaga.
- Wyjście: `quit`.

Ogólnie

- `set` ustawia różne parametry debuggera, `show` je wyświetla.
- `info` wyświetla informacje o śledzonym programie i różnych rzeczach, które podczas tego śledzenia ustawiamy (np. breakpointy).
- `help <polecenie>` pomaga.
- Wyjście: `quit`.
- W konsoli gdb działa uzupełnianie tabulacją.

Ogólnie

- `set` ustawia różne parametry debuggera, `show` je wyświetla.
- `info` wyświetla informacje o śledzonym programie i różnych rzeczach, które podczas tego śledzenia ustawiamy (np. breakpointy).
- `help <polecenie>` pomaga.
- Wyjście: `quit`.
- W konsoli `gdb` działa uzupełnianie tabulacją.
- Większość poleceń ma krótszą wersję, np. `backtrace` można wołać `bt`.

Ogólnie

- `set` ustawia różne parametry debuggera, `show` je wyświetla.
- `info` wyświetla informacje o śledzonym programie i różnych rzeczach, które podczas tego śledzenia ustawiamy (np. breakpointy).
- `help <polecenie>` pomaga.
- Wyjście: `quit`.
- W konsoli gdb działa uzupełnianie tabulacją.
- Większość poleceń ma krótszą wersję, np. `backtrace` można wołać `bt`.
- Jeśli chcemy rozpocząć od nowa uruchomienie, wystarczy użyć polecenia `run`.

Przebieg

- `stdout` i `stderr` śledzonego programu będzie widoczne w konsoli `gdb`.

Przebieg

- `stdout` i `stderr` śledzonego programu będzie widoczne w konsoli `gdb`.
- Program może normalnie zakończyć swoje działanie albo zostać przerwany z jakiegoś powodu.

Przebieg

- `stdout` i `stderr` śledzonego programu będzie widoczne w konsoli gdb.
- Program może normalnie zakończyć swoje działanie albo zostać przerwany z jakiegoś powodu.
- Powód przerwania jest wypisywany, zwykle jest to jakiś sygnał (np. `SIGSEGV`).

Przebieg

- `stdout` i `stderr` śledzonego programu będzie widoczne w konsoli `gdb`.
- Program może normalnie zakończyć swoje działanie albo zostać przerwany z jakiegoś powodu.
- Powód przerwania jest wypisywany, zwykle jest to jakiś sygnał (np. `SIGSEGV`).
- Można też klepnąć na klawiaturze `Ctrl+c`, czyli wysłać sygnał `SIGINT`.

Przebieg

- `stdout` i `stderr` śledzonego programu będzie widoczne w konsoli `gdb`.
- Program może normalnie zakończyć swoje działanie albo zostać przerwany z jakiegoś powodu.
- Powód przerwania jest wypisywany, zwykle jest to jakiś sygnał (np. `SIGSEGV`).
- Można też klepnąć na klawiaturze `Ctrl+c`, czyli wysłać sygnał `SIGINT`.
- Wznowienie działania zatrzymanego programu: `continue`.

Przebieg

- `stdout` i `stderr` śledzonego programu będzie widoczne w konsoli `gdb`.
- Program może normalnie zakończyć swoje działanie albo zostać przerwany z jakiegoś powodu.
- Powód przerwania jest wypisywany, zwykle jest to jakiś sygnał (np. `SIGSEGV`).
- Można też klepnąć na klawiaturze `Ctrl+c`, czyli wysłać sygnał `SIGINT`.
- Wznowienie działania zatrzymanego programu: `continue`.
- Powrót z bieżącej ramki stosu: `return`. Można w argumencie przekazać wartość, która ma być zwrócona z tej ramki.

Gdy już się wywali

...fajnie byłoby wiedzieć dlaczego.

```
Program received signal SIGSEGV, Segmentation fault.
0x00007fffe0000158 in ?? ()
```

Stos wywołań

```
(gdb) backtrace
#0  0x00007fffe0000158 in ?? ()
#1  0x000000000043d311 in ClientThread::cleanup (this=0x87a430)
    at .../src/networking/ClientThread.cpp:212
...
```

Stos wywołań

```
(gdb) backtrace
#0  0x00007fffe0000158 in ?? ()
#1  0x000000000043d311 in ClientThread::cleanup (this=0x87a430)
    at .../src/networking/ClientThread.cpp:212
...
```

W moim roboczym przykładzie jest dużo tekstu, który się tu nie zmieści, ale mogę pokazać pełen backtrace.

Stos wywołań

```
(gdb) backtrace
#0  0x00007fffe0000158 in ?? ()
#1  0x000000000043d311 in ClientThread::cleanup (this=0x87a430)
    at .../src/networking/ClientThread.cpp:212
...
```

W moim roboczym przykładzie jest dużo tekstu, który się tu nie zmieści, ale mogę pokazać pełen backtrace.

- Numerek po lewej (#0 i dalsze) to numer ramki stosu. Potem mamy adres w pamięci i informacja o miejscu wykonania (wraz z argumentami funkcji), która powinna być czytelna dla człowieka (a przynajmniej programisty).

Stos wywołań

```
(gdb) backtrace
#0  0x00007fffe0000158 in ?? ()
#1  0x000000000043d311 in ClientThread::cleanup (this=0x87a430)
    at .../src/networking/ClientThread.cpp:212
...
```

W moim roboczym przykładzie jest dużo tekstu, który się tu nie zmieści, ale mogę pokazać pełen backtrace.

- Numerek po lewej (#0 i dalsze) to numer ramki stosu. Potem mamy adres w pamięci i informacja o miejscu wykonania (wraz z argumentami funkcji), która powinna być czytelna dla człowieka (a przynajmniej programisty).
- Znaki zapytania się niefortunnie zdarzają – może nie mamy symboli debugowych na tym poziomie stosu, a może wykonanie programu poleciało w kosmos?

Jak widać ramka #0 jest nieużyteczna, ale przeskoczmy do kolejnej i zobaczmy na czym się wywala.

Jak widać ramka #0 jest nieużyteczna, ale przeskoczmy do kolejnej i zobaczmy na czym się wywala.

```
(gdb) frame 1
(gdb) list
207     }
208
209     void ClientThread::cleanup()
210     {
211             if (udpSocket != NULL) {
212                     udpSocket->close();
213                     delete udpSocket;
214             }
215     }
```

```
212                    udpSocket->close();
```

Ciekawe co jest nie tak z tym socketem.

```
212                         udpSocket->close();
```

Ciekawe co jest nie tak z tym socketem.

```
(gdb) print udpSocket
     (QUdpSocket *) 0x7fffe00013d0
```

```
212                          udpSocket->close();
```

Ciekawe co jest nie tak z tym socketem.

```
(gdb) print udpSocket
     (QUdpSocket *) 0x7fffe00013d0
```

To nie było zbyt pomocne. Ale widać, że gdb kuma typy z bibliotek.

```
212                         udpSocket->close();
```

Ciekawe co jest nie tak z tym socketem.

```
(gdb) print udpSocket
     (QUdpSocket *) 0x7fffe00013d0
```

To nie było zbyt pomocne. Ale widać, że gdb kuma typy z bibliotek.

```
(gdb) print *udpSocket
     <incomplete type>
```

```
212                          udpSocket->close();
```

Ciekawe co jest nie tak z tym socketem.

```
(gdb) print udpSocket
     (QUdpSocket *) 0x7fffe00013d0
```

To nie było zbyt pomocne. Ale widać, że gdb kuma typy z bibliotek.

```
(gdb) print *udpSocket
     <incomplete type>
```

...jak mamy dobry dzień.

Nie wszystkie typy są zdefiniowane w miejscu wywołania metody (forward declarations).

Nie wszystkie typy są zdefiniowane w miejscu wywołania metody (forward declarations).

```
(gdb) print this
     (ClientThread * const) 0x87a430
```

Nie wszystkie typy są zdefiniowane w miejscu wywołania metody (forward declarations).

```
(gdb) print this
     (ClientThread * const) 0x87a430
```

Przynajmniej moje typy kuma.

Nie wszystkie typy są zdefiniowane w miejscu wywołania metody (forward declarations).

```
(gdb) print this
     (ClientThread * const) 0x87a430
```

Przynajmniej moje typy kuma.

```
(gdb) print *this
     {<AbstractThread> = {<QThread> = {<No data fields>},
     static staticMetaObject = {d =
     {superdata = 0x659140 <QThread::staticMetaObject> ...
```

Zatrzymywanie ręczne

- `break` ustawia breakpoint na wskazanej linii kodu (format: `plik:numer`) albo funkcji.

```
(gdb) break ClientThread::cleanup()
Breakpoint 1 at 0x43d2e5:
    file .../src/networking/ClientThread.cpp, line 211.
```

Zatrzymywanie ręczne

- `break` ustawia breakpoint na wskazanej linii kodu (format: `plik:numer`) albo funkcji.

```
(gdb) break ClientThread::cleanup()
Breakpoint 1 at 0x43d2e5:
    file .../src/networking/ClientThread.cpp, line 211.
```

- `step` przenosi do następnej linii kodu.

Zatrzymywanie ręczne

- `break` ustawia breakpoint na wskazanej linii kodu (format: `plik:numer`) albo funkcji.

```
(gdb) break ClientThread::cleanup()
Breakpoint 1 at 0x43d2e5:
    file .../src/networking/ClientThread.cpp, line 211.
```

- `step` przenosi do następnej linii kodu.
- `stepi` przenosi do następnej instrukcji procesora. Przydatne przy grzebaniu w assemblerze.

Zatrzymywanie ręczne

- `break` ustawia breakpoint na wskazanej linii kodu (format: `plik:numer`) albo funkcji.

```
(gdb) break ClientThread::cleanup()
Breakpoint 1 at 0x43d2e5:
    file .../src/networking/ClientThread.cpp, line 211.
```

- `step` przenosi do następnej linii kodu.
- `stepi` przenosi do następnej instrukcji procesora. Przydatne przy grzebaniu w assemblerze.
- `skip` przeskakuje wywołania funkcji (nie zagłębia się w nie).

Zatrzymywanie ręczne

- `break` ustawia breakpoint na wskazanej linii kodu (format: `plik:numer`) albo funkcji.

```
(gdb) break ClientThread::cleanup()
Breakpoint 1 at 0x43d2e5:
    file .../src/networking/ClientThread.cpp, line 211.
```

- `step` przenosi do następnej linii kodu.
- `stepi` przenosi do następnej instrukcji procesora. Przydatne przy grzebaniu w assemblerze.
- `skip` przeskakuje wywołania funkcji (nie zagłębia się w nie).
- `print` już się pojawiło, a `continue` było wspomniane, ale to dobre miejsce na przypomnienie.

Zatrzymywanie ręczne

- `break` ustawia breakpoint na wskazanej linii kodu (format: `plik:numer`) albo funkcji.

```
(gdb) break ClientThread::cleanup()
Breakpoint 1 at 0x43d2e5:
    file .../src/networking/ClientThread.cpp, line 211.
```

- `step` przenosi do następnej linii kodu.
- `stepi` przenosi do następnej instrukcji procesora. Przydatne przy grzebaniu w assemblerze.
- `skip` przeskakuje wywołania funkcji (nie zagłębia się w nie).
- `print` już się pojawiło, a `continue` było wspomniane, ale to dobre miejsce na przypomnienie.
- `print` wypisuje podane wyrażenie raz, `display` wypisuje przy każdym zatrzymaniu programu.

Zatrzymywanie ręczne

- `break` ustawia breakpoint na wskazanej linii kodu (format: `plik:numer`) albo funkcji.

```
(gdb) break ClientThread::cleanup()
Breakpoint 1 at 0x43d2e5:
    file .../src/networking/ClientThread.cpp, line 211.
```

- `step` przenosi do następnej linii kodu.
- `stepi` przenosi do następnej instrukcji procesora. Przydatne przy grzebaniu w assemblerze.
- `skip` przeskakuje wywołania funkcji (nie zagłębia się w nie).
- `print` już się pojawiło, a `continue` było wspomniane, ale to dobre miejsce na przypomnienie.
- `print` wypisuje podane wyrażenie raz, `display` wypisuje przy każdym zatrzymaniu programu.
- Jest też `printf` do ładniejszego formatowania. Przydaje się, jeśli definiujemy własne funkcje.

Mój ~/.gdbinit z czasów programowania w assemblerze

```
set disassembly-flavor intel
set disassemble-next-line on

define pxd
        display $xmm8.v16_int8
        display $xmm7.v16_int8
        display $xmm6.v16_int8
        display $xmm5.v16_int8
        display $xmm4.v16_int8
        display $xmm3.v16_int8
        display $xmm2.v16_int8
        display $xmm1.v16_int8
        display $xmm0.v16_int8
end
```

Przerwania są bardzo silne i elastyczne. Kilka zachęcających linijek z `help breakpoints`:

- `watch` – Set a watchpoint for an expression
- Podobne: `awatch`, `rwatch`
- `commands` – Set commands to be executed when a breakpoint is hit
- `condition` – Specify breakpoint number N to break only if COND is true

Krwawe detale

- Własny debugger? `man 2 ptrace`

Krwawe detale

- Własny debugger? `man 2 ptrace`
- Pouczająca lektura, warto wiedzieć co potrafi ten syscall.

Krwawe detale

- Własny debugger? `man 2 ptrace`
- Pouczająca lektura, warto wiedzieć co potrafi ten syscall.
- Tak są robione np. ograniczone środowiska wywołana na konkursach programistycznych.

Krwawe detale

- Własny debugger? `man 2 ptrace`
- Pouczająca lektura, warto wiedzieć co potrafi ten syscall.
- Tak są robione np. ograniczone środowiska wywołana na konkursach programistycznych.
- Przydatne narzędzie śledzące program pod kątem używanych syscalli: `strace`.

Krwawe detale

- Własny debugger? `man 2 ptrace`
- Pouczająca lektura, warto wiedzieć co potrafi ten syscall.
- Tak są robione np. ograniczone środowiska wywołana na konkursach programistycznych.
- Przydatne narzędzie śledzące program pod kątem używanych syscalli: `strace`.
- Debugging gruboziarnisty, ale czasami szybko można namierzyć problem, np. zawieszenie na `select` albo `poll`.

Krwawe detale

- Własny debugger? `man 2 ptrace`
- Pouczająca lektura, warto wiedzieć co potrafi ten syscall.
- Tak są robione np. ograniczone środowiska wywołana na konkursach programistycznych.
- Przydatne narzędzie śledzące program pod kątem używanych syscalli: `strace`.
- Debugging gruboziarnisty, ale czasami szybko można namierzyć problem, np. zawieszenie na `select` albo `poll`.
- Nie trzeba programu kompilować jakoś szczególnie (nie musi być -ggdb).

Co ułatwia gdb?

- Śledzenie i badanie programu: breakpointy są mocne.

Co ułatwia gdb?

- Śledzenie i badanie programu: breakpointy są mocne.
- Szybkie łapanie głupich błędów (null-pointer dereference, off-by-one), o ile te błędy wcześnie powodują wysypkę zamiast mazać po pamięci.

Co ułatwia gdb?

- Śledzenie i badanie programu: breakpointy są mocne.
- Szybkie łapanie głupich błędów (null-pointer dereference, off-by-one), o ile te błędy wcześnie powodują wysypkę zamiast mazać po pamięci.
- . . . coś jeszcze?

Na co nie pomaga gdb?

- Heisenbugs.

Na co nie pomaga gdb?

- Heisenbugs.
- Głupotę/nieznajomość języka/nieuważność (undefined behavior).

Na co nie pomaga gdb?

- Heisenbugs.
- Głupotę/nieznajomość języka/nieuważność (undefined behavior).
- Nie jest kryształową kulą, która zastąpi asercje i logowanie ostrzeżeń, błędów czy zwykłych informacji diagnostycznych.