GDB

Rulare gdb

GDB poate fi folosit în două moduri pentru a depana programul:

- rulându-l folosind comanda gdb
- folosind fişierul core generat în urma unei erori grave (de obicei segmentation fault)

Cea de a doua modalitate este utilă în cazul în care bug-ul nu a fost corectat înainte de lansarea programului. În acest caz, dacă utilizatorul întâlneşte o eroare gravă, poate trimite programatorului fişierul core cu care acesta poate depana programul şi corecta bug-ul.

Cea mai simplă formă de depanare cu ajutorul GDB este cea în care dorim să determinăm linia programului la care s-a produs eroarea. Pentru exemplificare considerăm următorul program:

1 add.c

```
#include <stdio.h>

int f(int a, int b)
{
        int c;
        c = a + b;
        return c;
}

int main()
{
        char *bug = 0;
        *bug = f(1, 2);
        return 0;
}
```

După compilarea programului acesta poate fi depanat folosind GDB. După încărcarea programului de depanat, GDB intră în mod interactiv. Utilizatorul poate folosi apoi comenzi pentru a depana programul:

```
$ gcc -Wall -g add.c
$ gdb a.out
[...]
(gdb) run
Starting program: a.out

Program received signal SIGSEGV, Segmentation fault.
0x08048411 in main () at add.c:13
13              *bug=f(1, 2);
(gdb)
```

Prima comandă folosită este `run`. Această comandă va porni execuţia programului. Dacă această comandă primeşte argumente de la utilizator, acestea vor fi transmise programului. Înainte de a trece la prezentarea unor comenzi de bază din `gdb`, să demonstrăm cum se poate depana un program cu ajutorul fişierului core:

```
# ulimit -c 4
# ./a.out
Segmentation fault (core dumped)
# gdb a.out core
Core was generated by `./a.out'.
Program terminated with signal 11, Segmentation fault.
#0  0x08048411 in main () at add.c:13
13              *bug=f(1, 2);
(gdb)
```

Comenzi de bază GDB

Câteva din comenzile de bază în `gdb` sunt:

- **breakpoint** - primeşte ca argument un nume de funcţie (ex: main), un număr de linie şi, eventual, un fişier (ex: break sursa.c:50) sau o adresă (ex: break *0x80483d3).

- **next** - va continua execuţia programului până ce se va ajunge la următoarea linie din codul sursă. Dacă linia de executat conţine un apel de funcţie, funcţia se va executa complet.
- **step** - dacă se doreşte şi inspectarea funcţiilor.

Folosirea acestor comenzi este exemplificată mai jos:

```
$ gdb a.out
(gdb) break main
Breakpoint 1 at 0x80483f6: file add.c, line 12.
(gdb) run
Starting program: a.out

Breakpoint 1, main () at add.c:12
12              char *bug=0;
(gdb) next
13              *bug=f(1, 2);
(gdb) next

Program received signal SIGSEGV, Segmentation fault.
0x08048411 in main () at add.c:13
13              *bug=f(1, 2);
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
```

```
Starting program: a.out
Breakpoint 1, main () at add.c:12
12              char *bug=0;
(gdb) next
13              *bug=f(1, 2);
(gdb) step
f (a=1, b=2) at add.c:8
6               c=a+b;
(gdb) next
7               return c;
(gdb) next
8       }
(gdb) next

Program received signal SIGSEGV, Segmentation fault.
0x08048411 in main () at add.c:13
13              *bug=f(1, 2);
(gdb)
```

- **list** - această comandă va lista fişierul sursă al programului depanat. Comanda primeşte ca argument un număr de linie (eventual nume fişier), o funcţie sau o adresă de la care să listeze. Al doilea argument este opţional şi precizează câte linii vor fi afişate. În cazul în care comanda nu are nici un parametru, ea va lista de unde s-a oprit ultima afişare.
- **continue** - se foloseşte atunci când se doreşte continuarea execuţiei programului.

```
$ gdb a.out
(gdb) list add.c:1
1       #include <stdio.h>
2
3       int f(int a, int b)
4       {
5               int c;
6               c=a+b;
7               return c;
8       }
(gdb) break add.c:6
Breakpoint 1 at 0x80483d6: file add.c, line 6.
```

```
(gdb) run
Starting program: a.out

Breakpoint 1, f (a=1, b=2) at add.c:6
6               c=a+b;
(gdb) next
7               return c;
(gdb) continue
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x08048411 in main () at add.c:13
13              *bug=f(1, 2);
```

- **print** - cu ajutorul acesteia se pot afişa valorile variabilelor din funcţia curentă sau a variabilelor globale. `print` poate primi ca argument şi expresii complicate (dereferenţieri de pointeri, referenţieri ale variabilelor, expresii aritmetice, aproape orice expresie C validă). În plus, `print` poate afişa structuri de date precum `struct` şi `union`.

```
$ gdb a.out
(gdb) break f
Breakpoint 1 at 0x80483d6: file add.c, line 6.
(gdb) run
Starting program: a.out

Breakpoint 1, f (a=1, b=2) at add.c:6
6               c=a+b;
(gdb) print a
 class="code bash" = 1
(gdb) print b
$ gdb a.out (gdb) break f Breakpoint 1 at
0x80483d6: file add.c, line 6. (gdb) run
Starting program: a.out   Breakpoint 1, f (a=1,
b=2) at add.c:6 6 c=a+b; (gdb) print a $1 = 1
(gdb) print b $2 = 2 (gdb) print c $3 =
1073792080 (gdb) next 7 return c; (gdb) print c
$4 = 3 (gdb) finish Run till exit from #0 f
(a=1, b=2) at add.c:7 0x08048409 in main () at
add.c:13 13 *bug=f(1, 2); Value returned is $5
= 3 (gdb) print bug $6 = 0x0 = 2
(gdb) print c
 = 1073792080
(gdb) next
7               return c;
(gdb) print c
 = 3
(gdb) finish
Run till exit from #0  f (a=1, b=2) at add.c:7
0x08048409 in main () at add.c:13
13              *bug=f(1, 2);
Value returned is  = 3
(gdb) print bug
 = 0x0
```

```
(gdb) print (struct sigaction)bug
 = {__sigaction_handler =
 {
    sa_handler = 0x8049590 <object.2>,
    sa_sigaction = 0x8049590 <object.2>
 },
 sa_mask =
 {
    __val =
  {
    3221223384, 1073992320, 1, 3221223428,
    3221223436, 134513290, 134513760, 0, 3221223384,
    1073992298, 0, 3221223436, 1075157952,
    1073827112, 1, 134513360, 0, 134513393, 134513648, 1,
    3221223428, 134513268, 134513760, 1073794080,
    3221223420, 1073828556, 1, 3221223760, 0,
    3221223804, 3221223846,     3221223866
  }
 },
 sa_flags = -1073743402,
 sa_restorer = 0xbffff9f2}
(gdb)
```

Depanarea unui proces

Pe majoritatea sistemelor de operare pe care a fost portat, `gdb` nu poate detecta când un proces realizează o operaţie `fork`. Atunci când programul este pornit, depanarea are loc exclusiv în procesul **iniţial**, procesele copil nefiind ataşate debugger-ului. În acest caz, singura soluţie este introducerea unor întârzieri în execuţia procesului nou creat (de exemplu, prin apelul de sistem `sleep`), care să ofere programatorului suficient timp pentru a ataşa, manual, `gdb`-ul la respectivul proces, presupunând că i-a aflat `PID`-ul, în prealabil.

Pentru a ataşa debugger-ul la un proces deja existent, se foloseşte comanda `attach`, în felul următor:

```
(gdb) attach PID
```

Această metodă este destul de incomodă şi poate cauza chiar o funcţionare anormală a aplicaţiei depanate, în cazul în care necesităţile de sincronizare între procese sunt stricte (de exemplu operaţii cu timeout).

Din fericire, pe un număr limitat de sisteme, printre care şi Linux, `gdb` permite depanarea comodă a programelor care creează mai multe procese prin `fork` şi `vfork`. Pentru ca `gdb` să urmărească activitatea proceselor create ulterior, se poate folosi comanda `set follow-fork-mode`, în felul următor:

```
(gdb) set follow-fork-mode mode
```

unde `mode` poate lua valoarea `parent`, caz în care debugger-ul continuă depanarea procesului părinte, sau valoarea `child`, şi atunci noul proces creat va fi depanat în continuare. Se poate observa că în această manieră debugger-ul este ataşat la un moment dat doar la un **singur** proces, neputând urmări mai multe simultan.

Cu toate acestea, `gdb` poate ţine evidenţa tuturor proceselor create de către programul depanat, deşi, în continuare numai un singur proces poate fi rulat prin debugger la un moment dat. Comanda `set detach-on-fork` realizează acest lucru:

```
(gdb) set detach-on-fork mode
```

unde `mode` poate fi `on`, atunci când `gdb` se va ataşa unui singur proces la un moment dat (comportament implicit), sau `off`, caz în care `gdb` se ataşează la toate procesele create în timpul execuţiei, şi le suspendă pe acelea care nu sunt urmărite, în funcţie de valoarea setării `follow-fork-mode`.

Comanda `info forks` afişează informaţii legate de toate procesele aflate sub controlul `gdb` la un moment dat:

```
(gdb) info forks
```

De asemenea, comanda `fork` poate fi utilizată pentru a seta unul din procesele din listă drept cel activ (care este urmărit de debugger).

```
(gdb) fork fork-id
```

unde `fork-id` este identificatorul asociat procesului, aşa cum apare în lista afişată de comanda `info forks`.

Atunci când un anumit proces nu mai trebuie urmărit, el poate fi înlaturat din listă folosind comenzile `detach fork` şi `delete fork`:

```
(gdb) detach fork fork-id
(gdb) delete fork fork-id
```

Diferenţa dintre cele două comenzi este că `detach fork` lasă procesul să ruleze independent, în continuare, în timp ce `delete fork` îl încheie.

Pentru a ilustra aceste comenzi într-un exemplu concret, să considerăm programul următor:

forktest.c

```
 1      #include <stdio.h>
 2      #include <sys/types.h>
 3      #include <sys/wait.h>
 4      #include <unistd.h>
 5
 6
 7      int main(int argc, char **argv) {
 8          pid_t childPID = fork();
 9
10          if (childPID < 0) {
11              // An error occured
12              fprintf(stderr, "Could not fork!\n");
13              return -1;
14          } else if (childPID == 0) {
15
16              // We are in the child process
17                <a href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("The
child process is executing...\n");
18              sleep(2);
19
20          } else {
21
22              // We are in the parent process
23              if (wait(NULL) < 0) {
24                  fprintf(stderr, "Could not wait for child!\n");
25                  return -1;
26              }
2 7                                                                                                    <a
href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("Everything is done!\n");
28
29          }
30
31          return 0;
32      }
```

Dacă vom rula programul cu parametrii impliciţi de depanare, vom constata că `gdb` va urmări exclusiv execuţia procesului părinte:

```
$ gcc -O0 -g3 -o forktest forktest.c
$ gdb ./forktest
[...]
(gdb) run
Starting program: /home/mihnea/forktest
The child process is executing...
Everything is done!

Program exited normally.
```

Punem câte un `breakpoint` în codul asociat procesului părinte, respectiv procesului copil, pentru a evidenţia mai bine acest comportament:

```
(gdb) break 17
Breakpoint 1 at 0x8048497: file forktest.c, line 17.
(gdb) break 27
```

```
Breakpoint 2 at 0x80484f0: file forktest.c, line 27.

(gdb) run
Starting program: /home/mihnea/forktest
The child process is executing...

Breakpoint 2, main () at forktest.c:27
27                      printf("Everything is done!\n");
(gdb) continue
Continuing.
Everything is done!

Program exited normally.
```

Setăm debugger-ul să urmărească procesele copil, şi observăm că, de data aceasta, celălalt `breakpoint` este atins:

```
(gdb) set follow-fork-mode child
(gdb) run
Starting program: /home/mihnea/forktest
[Switching to process 6217]

Breakpoint 1, main () at forktest.c:17
17                      printf("The child process is executing...\n");
(gdb) continue
Continuing.
The child process is executing...

Program exited normally.
Everything is done!
```

Observaţi că ultimele două mesaje au fost **inversate**, faţă de cazul precedent: debugger-ul încheie procesul copil, apoi procesul părinte afişează mesajul de final (`Everything is done!`).

Nice to read

- 8 gdb tricks you should know