UNIWERSYTET ZIELONOGÓRSKI	Wydział Elektrotechniki, Informatyki i Telekomunikacji Instytut Informatyki i Elektroniki	

Instrukcja do zajęć laboratoryjnych

Język ANSI C (w systemie LINUX)

wersja: 1.34

Nr ćwiczenia:	3	
Temat:	**Debuger w systemie LINUX**	
Cel ćwiczenia:	Celem ćwiczenia jest zapoznanie się z zasadą debugowania programów w środowisku LINUX z wykorzystaniem programu gdb.	
Wymagane przygotowanie teoretyczne:	Znajomość podstaw programowania w języku C w systemie LINUX.	
Sposób zaliczenia:	Sprawozdanie w formie pisemnej. Pozytywna ocena ćwiczenia przez prowadzącego pod koniec zajęć.	[ ] [X]

1. Konwencje przyjęte w instrukcji

`Czcionka o stałej szerokości`

Nazwy programów, poleceń, katalogów, wyniki działania wydawanych poleceń.

`Czcionka o stałej szerokości pogrubiona`

Podaje tekst, który należy dosłownie przepisać. W przypadku plików źródłowych wyróżnia istotniejsze fragmenty.

`Czcionka o stałej szerokości kursywą`

Tekst komentarza w przykładowych sesjach przy terminalu.

`Czcionka o stałej szerokości kursywą pogrubiona`

Wyróżnia istotniejsze fragmenty wyników działania wydawanych poleceń.

2. Ćwiczenie

Poniżej pokazano bardzo prosty program, na bazie którego wykonane zostaną ćwiczenia z podstaw obsługi programu `gdb` (GNU debugger). Przepisując ten program należy oczywiście pominąć numery linii, które nie są częścią pliku źródłowego.

Celem przykładowego programu jest **próba nadpisania zabronionego obszaru pamięci**. Po uruchomieniu pojawia się komunikat: `Segmentation fault (core dumped)`.

Uwaga: aby na dysku utworzył się plik CORE trzeba wydać polecenie: `ulimit -c 10000`. Podana jako parametr liczba jest maksymalnym rozmiarem pliku CORE w KB - możemy ustawić ją dowolnie do potrzeb. Możemy też podać zamiast liczby słowo kluczowe `unlimited`.

Uwaga: administrator może zabronić użytkownikom ustawiać dowolnie dużą wartość pliku CORE. Zapoznajmy się z poniższym tekstem, który został przepisany z oryginalnej dokumentacji `GCC-HOWTO`:

```
Core files

   When Linux boots it is usually configured not to produce core files.
   If you like them, use your shell's builtin command to re-enable them:
   for C-shell compatibles (e.g. tcsh) this is
    % limit core unlimited

   while Bourne-like shells (sh, bash, zsh, pdksh) use
   $ ulimit -c unlimited
```

Uwaga: występujące w przykładach frazy typu `$xx` (gdzie `xx` jest pewną liczbą całkowitą) oznaczają kolejne lokalne zmienne, które powstają w wyniku wykonania kolejnych poleceń debugger'a. W twojej sesji liczby te najprawdopodobniej będą zupełnie inne.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define BIGNUM 5000
5
6  // prototypy funkcji
7  void fun_error (int tab[]);
8
9  int main (void)
10 {
11   int tab[10];
12   fun_error (tab);
13   exit (EXIT_SUCCESS);
14 }
15
16 void fun_error (int table[])
17 {
18   int i;
19   for (i = 0; i < BIGNUM; ++i)
20     table[i] = i;
21 }
```

Poniżej pokazano przykładową sesję przy komputerze. Pokazuje ona w zasadzie wszystkie te możliwości programu `gdb`, które pozwalają nam zlokalizować i usunąć błędy w oprogramowaniu. Celem nabrania wprawy należy powtórzyć na własnym komputerze tą sesję. Więcej informacji na temat programu `gdb` można znaleźć w dokumentacji – patrz spis literatury.

```
----------------------------- KOMPILACJA -----------------------------

Kompilujemy przykładowy program. Robimy to bez opcji -g. Rozmiar pliku
wynikowego wynosi 4883 bajty.
$ gcc debug_example.c -o debug_example
$ ls -l
total 12
-rwxr-xr-x    1 artur    users        4883 Apr  3 01:26 debug_example
-rw-r--r--    1 artur    users        1133 Apr  3 01:15 debug_example.c

Kompilujemy przykładowy program. Tym razem robimy to z opcją -g. Rozmiar
pliku wynikowego powiększa się do 17415 bajtów. Zawiera on dodatkowo
informacje przydatne do późniejszego debugowania.
$ gcc -g debug_example.c -o debug_example
$ ls -l
total 24
-rwxr-xr-x    1 artur    users       17415 Apr  3 01:26 debug_example
-rw-r--r--    1 artur    users        1133 Apr  3 01:15 debug_example.c

Poleceniem ulimit ustawiamy możliwość generowania plików core (tzw. pliki
rdzenia). Następnie wykonujemy program, który generuje błąd ochrony
pamięci i w efekcie kończy działanie zrzucając na dysk plik core.
Uwaga: administrator może zabronić użytkownikom ustawiać dowolnie dużą
wielkość pliku CORE. Wydając polecenie ulimit -a możemy dowiedzieć się
jakie mamy przyznane limity. Zwykli użytkownicy mogą te limity tylko
zmniejszać! Wówczas opcja unlimited oczywiście nie zadziała.
$ ulimit -c unlimited
$ ./debug_example
Segmentation fault (core dumped)

$ ls -l
total 76
-rw-------    1 artur    users       61440 Apr  3 01:27 core
-rwxr-xr-x    1 artur    users       17415 Apr  3 01:26 debug_example
-rw-r--r--    1 artur    users        1133 Apr  3 01:15 debug_example.c
```

```
--------------------- ROZPOCZĘCIE DEBUGOWANIA ---------------------------

Rozpoczynamy sesję debugowania. Używanie pliku rdzenia jest opcjonalne,
ale zwiększa możliwość debugowania. Opcja -q powoduje, że nie są
wyświetlane komunikaty licencyjne.
Analizując wynik widzimy, że program wygenerował sygnał 11, który oznacza
błąd segmentacji. Błąd pojawił się prawdopodobnie w linii 20 (wg. pliku
źródłowego) i on był powodem niespodziewanego zakończenia działania
programu.
$ gdb -q debug_example core
Core was generated by `_   `   a   b  '.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x08048440 in fun_error (table=0xbffffaa4) at debug_example.c:20
20          table[i] = i;
(gdb)

Z poziomu debugera uruchamiamy nasz program. Na ekranie widzimy komunikaty
w zasadzie identyczne z tymi zapisanymi w pliku core.
(gdb) run
```

```
Starting program: /home/artur/c/debug/debug_example

Program received signal SIGSEGV, Segmentation fault.
0x08048440 in fun_error (table=0xbffffaa4) at debug_example.c:20
20          table[i] = i;
(gdb)
```

```
---------------------- PRÓBA LOKALIZACJI BŁĘDU --------------------------

Wydając polecenie backtrace generujemy tzw. drzewo funkcji, które
doprowadziło do błędu segmentacji. Widzimy, że problem rozpoczął się w
funkcji main (linia 9) w momencie wywołania funkcji fun_error (w linii
12).
(gdb) backtrace
#0  0x08048440 in fun_error (table=0xbffffaa4) at debug_example.c:20
#1  0x08048402 in main () at debug_example.c:12
(gdb)

Używając funkcji list wyświetlamy te wiersze z kodu źródłowego, które
otaczają podejrzany fragment kodu (u nas ten w linii 20). Domyślnie
polecenie list wyświetla 10 wierszy. Można użyć polecenia list w takiej
formie: list n,m gdzie n i m są odpowiednio początkowym i końcowym numerem
wiersza, które chcemy wyświetlić.
(gdb) list 16,22
16      void fun_error (int table[])
17      {
18        int i;
19        for (i = 0; i < BIGNUM; ++i)
20          table[i] = i;
21      }
22
(gdb)
```

```
------------------ PODGLĄDANIE WARTOŚCI ZMIENNYCH ----------------------

Drukujemy bieżącą wartość zmiennej i. Widzimy, że program pracował do
momentu, gdy zmienna i osiągnęła wartość 343 (na twoim komputerze program
może załamać się w innym momencie).
(gdb) print i
$1 = 343

Dwa poniższe polecenia pokazują, że program nie ma dostępu do pamięci
począwszy od komórki 343 (to było powodem przedwczesnego zakończenia
działania programu), chociaż ma legalny dostęp do poprzedniej komórki
pamięci.
(gdb) print table[i]
Cannot access memory at address 0xc0000000
(gdb) print table[i-1]
$2 = 342
(gdb) print $1-1
$3 = 342

Poniższe polecenie NIE wyświetli wartości z tabeli table. Gdy program gdb
przyjrzy się elementowi table i zobaczy, że jest to adres pierwszego
elementu tablicy, to wyświetli ten adres i 9 następnych wartości jako
adresy pamięci. Użyteczność wyniku jest tu dyskusyjna.
```

Ogólna składnia polecenia: print tablica@numer, gdzie tablica jest nazwą tablicy lub obszaru pamięci, a numer jest liczbą wartości, które chcemy wyświetlić.

```
(gdb) print table@10
$4 = {0xbffffaa4, 0x0, 0x0, 0x8049590, 0x400097c0, 0x40130e58, 0x1,
  0x4011012c, 0x0, 0x1}
```

Wyświetlamy 5 wartości. Nawiasy kwadratowe wskazują, że chcemy rozpocząć od określonego miejsca w pamięci.

```
(gdb) print table[25]@5
$5 = {25, 26, 27, 28, 29}
```

Wyświetlamy 10 początkowych wartości (tablice w C są indeksowane od zera).

```
(gdb) print table[0]@10
$6 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Możliwe jest też formatowanie wydruku. Przykładowo argument polecenia print można sformatować jako liczbę dwójkową (jest to ciekawa opcja, gdyż w języku C nie ma kodu formatującego argument binarnie!). Inne dostępne kody – patrz dokumentacja gdb.

```
(gdb) print/x  table[0]@15
$7 = {0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8, 0x9, 0xa, 0xb, 0xc,
      0xd,  0xe}
(gdb) print/t  table[200]
$8 = 11001000
(gdb) print/t 4294967295    // 4294967295 = 2^32-1; maksymalna wartość int
$9 = 11111111111111111111111111111111
(gdb) print/t 4294967295+1  // 2^32
$10 = 0
```

Polecenie x umożliwia analizę zawartości pamięci. W pierwszej kolejności zapytamy się o adres 11-ego elementu (licząc od zera) tablicy table, a następnie wyświetlimy wartość 11-ego i 12-ego elementu tej tablicy w sposób pośredni, czyli poprzez adres.

(Uwaga: korzystając z konsoli PuTTY, można sobie ułatwić wprowadzanie długich elementów zaznaczając na ekranie myszką jakiś fragment, a następnie po ustawieniu kursora w pożądanym miejscu przeniesienie zaznaczonej wartości poprzez kliknięcie prawym przyciskiem myszki)

```
(gdb) print  table+10
$11 = (int *) 0xbffffadc
```

zaznacz to myszką ...

... i kliknij prawym klawiszem myszki

```
(gdb) x/2d 0xbffffadc      // /2d – wyświetl dwa elementy dziesiętnie
0xbffffadc:     10     11
(gdb)  x/2x 0xbffffadc     // /2x – wyświetl dwa elementy szesnastkowo
0xbffffadc:     0x0000000a      0x0000000b
```

Dostęp do pamięci jest kontrolowany. Przykładowo, wiemy z wcześniejszej analizy, że program zawiesił się przy wpisie do tablicy o indeksie 343. Zawartości pamięci poza tym „adresem" nie uda się wyświetlić.

```
(gdb) print i
$12 = 343
(gdb) print table+342
$13 = (int *) 0xbffffffc  // wartość typu (int*), czyli adres
(gdb) x/d 0xbffffffc
0xbffffffc:     342
(gdb) print table+343
$14 = (int *) 0xc0000000  // size(int) = 4; 0xbffffffc + 0x4 = 0xc0000000
(gdb) x/d 0xc0000000
0xc0000000:     Cannot access memory at address 0xc0000000
```

```
Za pomocą polecenia whatis możemy określić typ zmiennej lub funkcji.
(gdb) whatis i
type = int
(gdb) whatis table
type = int *
(gdb) whatis fun_error
type = void (int *)
(gdb)
```

```
---------------------- USTAWIANIE PUŁAPEK --------------------------

Składnia:
break numer_linii
break nazwa_funkcji
break nazwa_pliku:numer_linii
break nazwa_pliku:nazwa_funkcji
break numer_linii if wyrażenie    (pułapka warunkowa, bardzo użyteczne!)
break nazwa_funkcji if wyrażenie  (pułapka warunkowa, bardzo użyteczne!)

continue         - wznawianie wykonywania
delete           - kasowanie pułapek
info breakpoints - lista bieżących pułapek

Zakładamy pierwszą pułapkę (warunkową) w linii 20 i uruchamiamy program.
(gdb) break 20 if i == 15
Breakpoint 1 at 0x8048430: file debug_example.c, line 20.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/artur/c/debug/debug_example

Program przerwał działanie, gdy zmienna i osiągnęła wartość 15.
Breakpoint 1, fun_error (table=0xbffffaa4) at debug_example.c:20
20       table[i] = i;

Potwierdzamy, że zmienna i rzeczywiście ma wartość 15.
(gdb) print i
$8 = 15

Gdy zapomnimy gdzie i jakie pułapki są założone. Przypomni nam o tym
polecenie breakpoints.
(gdb) info breakpoints
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x08048430 in fun_error at debug_example.c:20
        stop only if i == 15
        breakpoint already hit 1 time
(gdb) delete 1
(gdb)

Zakładamy drugą pułapkę (warunkową) w linii 20 i uruchamiamy program.
(gdb) break 20 if i == 15
Breakpoint 2 at 0x8048430: file debug_example.c, line 20.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/artur/c/debug/debug_example

Breakpoint 2, fun_error (table=0xbffffaa4) at debug_example.c:20
20          table[i] = i;
```

```
(gdb) print i
$15 = 15

Zmieniamy „w locie” wartość zmiennej i. Nadajemy jej wartość 12.
(gdb) set variable i = 12
(gdb) print i
$16 = 12

Kontynuujemy wykonywanie programu (krokowo – polecenie step). Uwaga:
zamiast wpisywać za każdym razem słowo step możemy po prostu naciskać
ENTER, gdyż program gdb pamięta ostatnio wydane polecenie.
(gdb) step
19        for (i = 0; i < BIGNUM; ++i)
(gdb) step
20          table[i] = i;
(gdb) print i
$17 = 13
(gdb) step
19        for (i = 0; i < BIGNUM; ++i)
(gdb) step
20          table[i] = i;
(gdb) print i
$18 = 14
(gdb) step
19        for (i = 0; i < BIGNUM; ++i)
(gdb) step

Zmienna i ponownie osiągnęła wartość 15, więc uaktywniła się założona
wcześniej pułapka.
Breakpoint 2, fun_error (table=0xbffffaa4) at debug_example.c:20
20          table[i] = i;
(gdb) print i
$19 = 15
(gdb)

Kończymy prace z debugerem.
(gdb) q
A debugging session is active.
Do you still want to close the debugger?(y or n) y

Informacje dla debugera można usunąć z pliku debug_example bez
konieczności powtórnej rekompilacji programu bez użycia opcji -g.
Polecenie strip usuwa wszelkie symbole z dowolnego pliku (ang. strip –
discard symbols from object files).
$ ls -l
total 12
-rwxr-xr-x    1 artur    users        17415 Apr  3 01:26 debug_example
-rw-r--r--    1 artur    users        1133  Apr  3 01:15 debug_example.c

$ strip debug_example
$ ls -l
total 24
-rwxr-xr-x    1 artur    users        3056 Apr  3 01:26 debug_example
-rw-r--r--    1 artur    users        1133 Apr  3 01:15 debug_example.c

Zauważ, że wielkość pliku po zastosowaniu strip debug_example jest
mniejsza, niż po kompilacji gcc debug_example.c -o debug_example. Oznacza
to, że kompilator "samoistnie" dokłada do pliku wynikowego pewną ilość
symboli.
```

```
---------------------------- WSKAŹNIKI I ADRESY ------------------------

#include <stdio.h>
#include <stdlib.h>

int main (void)
{
  int   i = 0x81FF0F55;
  float f = 0.2;           // dwójkowo liczba 0.2 ma rozwinięcie okresowe!
  float *pf = &f;
  int   *pi = &i;
}
```

Uwaga: po uruchomieniu debugera i wydaniu polecenia **run**, *program poprawnie zakończy się (gdyż w odróżnieniu od przykładów używanych powyżej nie jest błędny). Aby więc poniższe polecenie* **print pi** *nie zakończyło się komunikatem:*

```
No symbol "pi" in current context.
```

należy ustawić punkt zatrzymania (breakpoint), na przykład na ostatnim nawiasie klamrowym }.

```
(gdb) list
1       #include <stdio.h>
2       #include <stdlib.h>
3       int main (void)
4       {
5               int i = 0x81FF0F55;
6               float f = 0.2;
7               float *pf = &f;
8               int *pi = &i;
9               return (0);
10      }
(gdb) break 8
Breakpoint 1 at 0x8048384: file debug2.c, line 8.
```

Teraz można już kontynuować ćwiczenie.

Pamiętajmy, że pi *to wskaźnik na zmienną* i. *W wyniku dostajemy więc adres zmiennej a nie jej wartość.*

```
(gdb) print pi
$2 = (int *) 0xbffffad8
```

Wyświetlamy wartość, na którą wskazuje wskaźnik (dzięki przełącznikowi /t *wynik jest w postaci dwójkowej).*

```
(gdb) x/t 0xbffffad8
0xbffffad8:     10000001111111110000111101010101
```

Wewnętrznie w rejestrach procesora liczba 4. bajtowa reprezentowana jest w postaci zwanej little endian *(sposób zapisu słów bajtowych, w którym pierwszeństwo ma bajt mniej znaczący. Mniej znaczący bajt występuje jako pierwszy).*
Teraz wyświetlamy 4 kolejne bajty pamięci poczynając od adresu 0xbffffad8.

```
 (gdb) x/4b 0xbffffad8
0xbffffad8:     01010101        00001111        11111111        10000001
(gdb)
```

Poniżej pokazano zasadę zapisu liczb w konwencji little endian.
Wartość zmiennej i:

```
10000001 11111111 00001111 01010101        // = 0x81FF0F55

Zamiana kolejności słów:
00001111 01010101 10000001 11111111

Zamiana kolejności bajtów w słowach:
01010101 00001111 11111111 10000001

Zobaczmy jak wygląda wewnętrzna reprezentacja liczby zmiennoprzecinkowej
0.2. W zapisie dwójkowym nie da się jej przedstawić dokładnie! Dlaczego?
(gdb) print pf
$1 = (float *) 0xbffffad4
(gdb) x/t 0xbffffad4
0xbffffad4:     00111110010011001100110011001101
                          ^^^^^^^^^^^^^^^^^^^^
                          tu widać okresowość

Dla liczb o pojedynczej precyzji (float) „niedoskonałość" zapisu widać już
na 9. miejscu po przecinku.
(gdb) print f
$3 = 0.200000003
(gdb) x/f 0xbffffad4
0xbffffad4:     0.200000003

Uwaga. Polecenie x wymaga argumentu (adresu pamięci). Niepodanie go,
spowoduje wyświetlenie wartości pamięci z przypadkowego adresu.
Prawdopodobnie będzie to następny adres w stosunku do poprzedniego
wywołania polecenia x, ale reguły tu nie ma. Na pewno jednak na ekranie
pojawi się „dziwna" wartość
(gdb) x/f
0xbffffad5:     -9.36942065e-38
```

3. Inne systemy

Warto zapoznać się też z nakładkami na gdb (bardzo ułatwiają pracę z debugerem) oraz kompilatorami klasy gcc przeniesionymi na platformę Windows (zawierają graficzne IDE):

cgdb (http://cgdb.sourceforge.net/)
dbvi (http://dbvi.sourceforge.net/)

lcc-win32 (http://www.cs.virginia.edu/~lcc-win32/) – darmowy kompilator („klasy gcc") dla Windows z bardzo funkcjonalnym interfejsem.
DevC++ (http://www.bloodshed.net/devcpp.html) – j.w.

Literatura

1. Polecenia systemowe `man gdb`, `info gdb`, `man strip`

2. *Using GDB: A Guide to the GNU Source-Level Debugger*, Richard M. Stallman i Roland H. Pesch, July 1991. Tenże tekst jest dostępny online jako wpis `gdb` programu `info`. Wydając polecenie `info gdb -o nazwa-pliku --subnodes` możemy cały ten tekst zapisać do pliku tekstowego.

3. *GDB Quick Reference*, dostępne na stronie WWW prowadzącego. Dwie strony krótkiej „ściagawki" z polecenia `gdb`