RUS Minikurs

Symbolisches Debuggen mit gdb und dbx

J. Scheurich ICA 2
scheurich@rus.uni-stuttgart.de

1. Anwendungsgebiet eines Symbolischen Debuggers

2. Compilierung

3. Aufruf

4. Die grundlegenden Befehle

5. Fehler, die zu einem Programmabsturz führen

6. Beispiel und kleinere Übungen

Anwendungsgebiet eines Symbolischen Debuggers

Debugger erlauben das Betrachten von Programm-abläufen und -zwischenergebnissen ohne den Quellcode zu verändern.

Am nützlichsten sind Debugger bei Programmabstürzen.

Unter Unix wir bei einem Programmabsturz ein 'core'-files angelegt.

Dieses 'core'-file enthält den Speicherinhalt des abgestürzten Programms.

Der Debugger kann aus dem 'core'-file die Absturz-Stelle und Variableninhalte ermitteln.

Debugger sind hier der konventionellen Methode der Fehlersuche durch verschiedene Programmausgaben überlegen, da unter Unix die Ausgabe wegen der Pufferung nicht immer wie gedacht arbeitet (Ein Beispiel dazu folgt später).

Debugger sind besonders nützlich, wenn die problematische Stelle ungefähr klar ist. Zum Beispiel wenn man aus dem 'core'-file die Absturz-Stelle ermitteln kann oder man sich sowieso nur für den Inhalt einer Variablen an einer bestimmten Stelle interessiert. Ansonsten ist der Rechenzeit- oder Tippaufwand unvernünftig hoch.

Compilierung

Der Debugger benötigt Informationen über Namen und Adresse der verwendeten Symbole (Variablen- und Funktionen-namen) sowie Referenzen zum Quellcode.

Diese Informationen wird während der Compilierung und Bindung des Programms erzeugt und im ausführbaren File abgespeichert.

Die Standartoption unter Unix zum **g**enerieren symbolischer Informationen ist -g

cc **-g** file.c (bzw. gcc **-g** file.c)

f77 **-g** file.f

pas **-g** file.p

g++ **-g** file.cc

gcc **-g** file.adb

usw.

Es wird dringend empfohlen, **nicht** die Option -O fuer **O**ptimierung zu benutzen. Nur wenige Compiler unterstützen die gleichzeitige Benutzung von **-g** und **-O** (u.U. gcc/g++), diese geben dies aber in der Manpage ausdrücklich bekannt.

Achtung: Nicht bei allen Implementierungen unterstützt der gcc/g++ Compiler die **-g** Option.

Bei der Verwendung von Translatoren, die von einer Programmiersprache in eine andere wandeln, kann in einigen Fällen die korrekte Generierung symbolischer Information erzwungen werden (bekannt ist das fuer f2c).

Aufruf

Bevor der Debugger aufgerufen wird, ist unter Umständen zuerst die Erzeugung eines 'core'-files von Interesse.

Die Erzeugung eines 'core'-files wird mit den shell-internen limit Befehlen in der ksh **ulimit -c** bzw. in der csh **limit coredumpsize** (siehe manpages der shells) gesteuert.

ulimit -c zeigt die maximal mögliche Länge eines 'core'-files. Mit **ulimit -c neue_länge** kann diese Länge gesetzt werden. Bei der Länge 0 wird das Schreiben eines 'core'-files unterdrückt.

Beim Aufruf selbst muss man zwischen den Debuggern **dbx** und **gdb** unterscheiden.

Für **dbx**:
dbx ausführbares_programm Das 'core'-file wird nötigenfalls automatisch dazugeladen.

Für **gdb**:
gdb ausführbares_programm core
oder
gdb ausführbares_programm
wenn kein 'core'-file vorhanden ist.

dbx hat in der Regel zusätzlich eine Option, mit der auf einen laufenden Prozess mit einer bestimmen Prozessnummer (PID) zugegriffen werden kann. Neuere Versionen von gdb erlauben die Angabe der PID anstatt des corefiles. Damit ist man in der Lage, auch parallelisierte Programme zu debuggen.

Es gibt für diese Debugger auch graphische Implementationen
dbx : xdbx, xde (AIX), cvd/edge (IRIX), ddd
gdb : xxgdb, ddd

gdb wurde für den gcc/g++ -Compiler entwickelt, **dbx** für die Compiler des jeweiligen Computerherstellers. Es ist derher nicht gesagt, daß die Debugger etwas mit dem Output des anderen Compilers anfangen können.

gdb bietet derzeit keine echte Sprachunterstützung für andere Sprachen als **C** oder **C++**.
Manche **dbx** Versionen bietet immer noch keine echte Sprachunterstützung für **C++**.

Die grundlegenden Befehle

Bis auf ein paar Ausnahmen heissen die einfachsten Befehle in **dbx** und **gdb** gleich.

Mit dem Kommando **help** kann interaktiv die genaue Syntax eines Befehls ermittelt werden. Im Rahmen dieses Minikurses wird nur die wichtigste Anwendung eines Befehls besprochen.

Alle Befehle können eindeutig abgekürzt werden.

In **dbx** kann man das Sourcefile mit dem **file sourcefile** Befehl wechseln. Zeilennummern können mit **gdb** auch in der Form **sourcefile:zeilennummer** verwendet werden.

list argument dient zur Orientierung und listet ein Stück des Quellcodes. Mögliche Argumente sind eine Zeilennummer oder der Name eines Unterprogramms. Ohne Argument wird das nächste Stück gelistet.

Um das Programm im Debugger zu starten wird
run kommandozeilenargumente verwendet.

Der Befehl **where** dient zur Ausgabe der Stelle, an dem das Programm gestoppt wurde (z.B. bei einem Programmabsturz). Bei **gdb** wird immer die gesamte Unterprogrammsvorgeschichte einschließlich der der Argumente ausgegeben. Bei **dbx** muss der Befehl u. U. in der Form
where zahl_unterprogramme angeben werden, um den gleichen Effekt zu erreichen.

Mit **print argument** werden Adressen, Variablen, Zeiger, Speicherinhalte oder sogar größere Ausdrücke (z.B. arrays/structs) ausgegeben. Zusätzlich dazu ergibt der Operator **&** die Adresse einer Variable, der Operator ***** den Inhalt einer Adresse (z.B. print * 0x00b4). Die Debugger **dbx** und **gdb** verstehen in der Regel immer die Syntax der Sprache **C**.

Der **dbx**-Befehl **trace** bewirkt, daß alle ausgeführten Befehle (z.B. nach **run**) ausgegeben werden. Für diesen Befehl gibt es in dieser Form keine Entsprechung in **gdb**. Neuere Versionen von **dbx** akzeptieren diesen Befehl nicht mehr.

Der **dbx**-Befehl in der Form **trace variablenname** gibt jede Änderung der Variablen gemeldet. Die **gdb** Entsprechung dazu heißt **watch variablenname**. Häufig wird dabei ein sogenannter Hardware-Watchpoint unterstützt. Dabei wird jede Änderung der entsprechenden Speicheradresse gemeldet, auch wenn die dazugehörende Programmzeile scheinbar nichts mit dieser Adresse zu tun haben scheint (beispielsweise wenn man über die Grenzen eines Arrays hinausliest, und dabei den Speicherbereich der nächsten Variablen erwischt). Wenn ein System keinen Hardwarewatchpoint zur Verfügung stellt, wird diese Art des Debuggens sehr zeitaufwendig, und sollte wenn möglich nur lokal eingesetzt werden.

Um einen Breakpoint zu setzen, bei dem ein Programmlauf gestoppt werden kann, wird mit **gdb** der Befehl **break** und mit **dbx** der Befehl **stop** benutzt. **dbx** unterscheidet zusaetzlich zwischen **stop at zeilennummer** und
stop in routine.

Um nach Erreichen eines Breakpoints den Programmlauf fortzusetzen, wird **cont** benutzt.

Um nach Erreichen eines Breakpoints die nächste Zeile auszuführen gibt es den Befehl **next**.

Um nach Erreichen eines Breakpoints den nächsten Programmbefehl (der auch in ein Unterprogramm führen kann) auszuführen gibt es den Befehl **step**.

Der Befehl für die Ausgabe einer Liste der Break- und Watchpoints in **dbx** heisst **status**, während dafür in **gdb** dafür der Befehl **info breakpoints** benutzt wird. In dieser Liste sind Break- und Watchpoints durchnummeriert.

Mit dem Befehl **delete nummer_der_liste** kann der entsprechende Break- bzw. Watchpoint aus der Liste gelöscht werden.

Um einen Breakpoint oder Watchpoint nur unter einer bestimmten Bedingung aktiv werden zu lassen, wird in **dbx** der Ausdruck **if logischer_ausdruck** an den **stop** oder **trace** Befehl angehängt (z.B. stop at 19 if x == 3).
Bei **gdb** wird ein extra Befehl
condtion nummer_der_liste logischer_ausdruck benötigt (z.B. condition 1 i==3).

Mit **quit** wird der Debugger verlassen.

	dbx	gdb
Interaktive Hilfe	help	help
Quellcode anschauen	list	list
Variablen ausgeben	print	print
Programm neu starten	run	run
Programmablauf beobachten	trace	
Aufrufbaum betrachten	where	where
Breakpoint setzen	stop at	break
	stop in	break
Programmlauf fortsetzen	cont	cont
Nächste Programmzeile	next	next
Nächster Programmbefehl	step	step
Break/Watchpointliste	status	info breakpoints
Bedingter Break/Watchpoint	befehl if	befehl condition
Debugger verlassen	quit	quit

Fehler, die zu einem Programmabsturz führen

Floating point exception
Tritt auf bei mathematischen Operationen, bei denen ein Ergebnis nicht errechnet werden kann: negative Wurzel, negativer Logarithmus usw.

Segmentation violation
Das Programm versucht auf Speicherbereiche zuzugreifen, die nicht ihm gehören. Das kann passieren, wenn über inkorrekte Zeiger zugegriffen wird, wenn Arraygrenzen stark überschritten werden, versucht wird eine Konstante zu überschreiben, oder wenn die Zahl der Aufrufargumente beim Unterprogrammaufruf nicht stimmt.

Memory fault
Es ist ein Fehler bei der Speicherorganisation aufgetreten . Dies wird zum Beispiel durch einen Aufruf der free()-funktion mit einem Pointer, der nicht auf einen Bereich zeigt, der mit alloc() angefordert wurde, erreicht.

Bus error
Eine unmögliche Anforderung an den Speicherbus trat auf.
Wichtig ist in diesem Zusammenhang der sogenannte Alignfehler.
Einige Prozessoren (vor allem RISC-Prozessoren) können wegen ihrer Busorganisation beim Speicherzugriff auf mehrere Bytes gleichzeitig (typischerweise beim Zugriff auf einen 4 Byte langen Integer/Floatwert) nur zugreifen, wenn die Adresse ein mehrfaches der Anzahl der Bytes ist.

Illegal instruction
Das Programm wurde an einer Stelle abgebrochen, an der ein unbekannter Maschinenbefehl aufgetreten ist. Das kann durch den falschen Aufruf von longjmp() oder einem Compilerfehler passiert sein.

Programmabstürze sind nicht portabel.

Zum Beispiel führen nicht alle Architekturen per default eine **floating point exception** bei einer Division durch Null durch.
Eine unterschiedliche Speicherorganisation bei verschiedenen Systemen kann zu unterschiedlichen Verhalten bei **Segmentation violation** und **Memory fault** führen.
Verschiedenes Alignverhalten kann ebenfalls zu Problemen bei der Portierung von Programmen führen.

Beispiele

```
$ulimit -c
unlimited
$f77 -g test.f
$a.out
%1     493 Segmentation violation - core dumped a.out
$gdb a.out core

GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.10.pl1 (i486-unknown-linux),
Copyright 1993 Free Software Foundation, Inc...
(gdb) list
1 program nix
2 character *10 i,j
3         i='1'
4         j='2'
5         call swap(i,j)
6         call swap(j,'3')
7         end
8         subroutine swap(i,j)
9         character*(*) i,j
10         character*(100) temp
(gdb) list
11         temp=i
12         i=j
13         j=temp
14         end
#0  0x2ba in s_copy ()
(gdb) where
#0  0x2ba in s_copy ()
#1  0x10a in swap_ (i=0x118ac "1         ", j=0x54 "3", i_len=10, j_len=1)
    at test.f:13
#2  0xbc in MAIN__ () at test.f:6
#3  0x293 in main ()
(gdb) break 2
Breakpoint 1 at 0x64: file test.f, line 2.
```

```
(gdb) run
Starting program: /home/mufti/f/a.out

Breakpoint 1, MAIN__ () at test.f:2
2 character *10 i,j
(gdb) print i
$1 = "\000\000\000\000\000\000\000\000\000"
(gdb) break test.f:5
Breakpoint 2 at 0x93: file test.f, line 5.
(gdb) continue
Continuing.

Breakpoint 2, MAIN__ () at test.f:5
5         call swap(i,j)
(gdb) print i
$2 = "1         "
(gdb) print j
$3 = "2         "
(gdb) next
6         call swap(j,'3')
(gdb) print i
$4 = "2         "
(gdb) print j
$5 = "1         "
(gdb) list test.f:1
1 program nix
2 character *10 i,j
3         i='1'
4         j='2'
5         call swap(i,j)
6         call swap(j,'3')
7         end
8         subroutine swap(i,j)
9         character*(*) i,j
10         character*(100) temp
(gdb) step
swap_ (i=0x118ac "1         ", j=0x54 "3", i_len=10, j_len=1) at test.f:11
11         temp=i
(gdb) print i
$6 = 0x118ac "1         "
(gdb) next
12         i=j
(gdb) next
13         j=temp
(gdb) next
```

Beispiel zur Pufferung (Linux 0.99, bash, gcc, gdb)

```
/home/mufti/c$ bash
/home/mufti/c$ ulimit -c
unlimited
/home/mufti/c$ cat pufferfehler.c
#include <stdio.h>

main()
{
int i;
for (i=1;i<=ZAHL;i++)
   printf("%d ",i);
i=i/0;
printf("%d\n",i);
}
/home/mufti/c$ gcc -g -DZAHL=283 pufferfehler.c
/home/mufti/c$ a.out
Floating point exception (core dumped)
/home/mufti/c$ gcc -g -DZAHL=284 pufferfehler.c
/home/mufti/c$ a.out
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 5
7 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83
84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107
108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127
128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147
148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167
168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187
188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207
208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227
228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247
248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267
268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 Floating point e
xception (core dumped)
/home/mufti/c$ gdb a.out core
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.10.pl1 (i486-unknown-linux),
Copyright 1993 Free Software Foundation, Inc...
#0  0xac in main () at pufferfehler.c:8
8 i=i/0;
(gdb) print i
$1 = 285
(gdb)
```