DEBUGGING ON LINUX

Shakti D Shekar *VJTI, Mumbai*
Prof. Dr. B B Meshram *VJTI, Mumbai*
Prof. Varshapriya *VJTI, Mumbai*
Pranit Patil *VJTI, Mumbai*
Pranav Ambavkar *VJTI, Mumbai*

ABSTRACT
In programming sometimes some condition which was assumed to be true is false during the program execution or the value of the variable which was assumed is not during program execution. The result of this the output will not be as expected. The errors like segmentation fault, logical errors can be understood during program execution only. Therefore debugging is very important in programming. The debugging can be done by printing out message or by using debugging tools or by using manual checking of the code line by line. There are various tools available to test the program for debugging and some of them are specific error debuggers. In this paper we discuss various debugging tools available for C programming on Linux platform. These tools helps programmer to reduce time in manual checking of program line by line.

***Keywords*- GDB, MEMWATCH, strace, valgrind.**

I. INTRODUCTION
The debugging is process of finding and reducing number of bugs in a program to make it behave as expected. In the high level programming language like Java have feature of exception handling that makes debugging easier. But programming language like C may cause silent problem such as memory corruption and we cannot test program run time. The debuggers are software tools which can debug the program line by line or from specific line number by setting break points or by changing values in memory. In this paper we did literature survey of four tools Valgrind, MEMWATCH, strace and GDB. For each tool we have tested programs on Ubuntu 11.10 machine having kernel 2.6.39.4. The paper is organized as follows: section 2 deals with Valgrind, section 3 deals with MEMWATCH, section 4 deals with strace utility, and section 5 deals with GDB. Finally we conclude in section 6.

II. Valgrind
The Valgrind[1][3] is for finding memory management problems and threading bugs. It includes memory error detector, two thread error detectors, a cache and branch-prediction profiler, a call graph generating cache and branch prediction profiler, and a heap profiler. This is developed by Julian Seward and ported to the Power architecture by Paul Mackerras. Valgrind can be used to develop new tool.

The memory leak is problem caused when program reserves the memory but it does not release back to the operating system. If the program continuously reserving memory without releasing it then system might turn into crash. As a developer sometimes it is very difficult to find out manually to detect exact cause of the error and line at which error could be if code is in thousands of lines. In such case best option is to test the program for debugging using valgrind. Here is the program file test1.c (fig. 1) where memory leakage is the problem.

```
#include<stdio.h>
#include<stdlib.h>
main()
{
  char *p1;
  p1=malloc(10);
  return 0;
}
```

Fig 1: test1.c

Compile test1.c with –g option for debugging and use valgrind with -v option to show output on console. Fig. 2 shows last few lines from valgrind output. In the output leak summary is given where we understood problem with the program is memory leakage. The valgrind detects invalid pointer use, use of uninitialized variables too. The program code test2.c (Fig. 3) is used for such errors. Fig. 4 shows last few lines from the output of valgrind for such buggy program.

```
sony@sony-VPCEB14EN:~/debug$ gcc -g test1.c
-o test1op
sony@sony-VPCEB14EN:~/debug$ valgrind -v
./test1op
…
==1935== Searching for pointers to 1 not-freed
blocks
==1935== Checked 77,912 bytes
==1935==
==1935== LEAK SUMMARY:
==1935==    definitely lost: 10 bytes in 1 blocks
==1935==    indirectly lost: 0 bytes in 0 blocks
==1935==      possibly lost: 0 bytes in 0 blocks
==1935==    still reachable: 0 bytes in 0 blocks
==1935==         suppressed: 0 bytes in 0 blocks
==1935== Rerun with --leak-check=full to see
details of leaked memory
==1935==
==1935== ERROR SUMMARY: 0 errors from 0
contexts (suppressed: 4 from 4)
--1935--
--1935-- used_suppression:      4 dl-hack3-cond-1
==1935==
==1935== ERROR SUMMARY: 0 errors from 0
contexts (suppressed: 4 from 4)
```

Fig. 2 Valgrind output of test1.c

From output in the fig. 4 we understood that there are 3 bugs. First (at line no. 15) conditional jump depends on uninitialized value this is because of we were trying to free unreserved memory. Second (at line no. 12), invalid write of size 1, because we were accessing p[2] which is invalid. Third (at line no. 7), conditional jump depends on uninitialized value because variable i hadn't initialized and we were accessing it. However valgrind does not check bound checking on static arrays. That's why it haven't shown an error at line no. 13 which is a[5]=b.

```
#include<stdio.h>
#include<stdlib.h>
main()
{
  int i;
  char *p, *q, a[5];
  if(i==0)
  {
  printf("unreachable");
  }
  p=malloc(2);
  p[2]='a';
  a[5]='b';
  free(p);
  free(q);
  return 0;
}
```

```
…
==2028== ERROR SUMMARY: 3 errors from 3
contexts (suppressed: 4 from 4)
==2028==
==2028== 1 errors in context 1 of 3:
==2028== Conditional jump or move depends on
uninitialised value(s)
==2028==    at 0x4C28293: free
(vg_replace_malloc.c:366)
==2028==    by 0x400657: main (test2.c:15)
==2028==
==2028==
==2028== 1 errors in context 2 of 3:
==2028== Invalid write of size 1
==2028==    at 0x400639: main (test2.c:12)
==2028==  Address 0x51ce042 is 0 bytes after a
block of size 2 alloc'd
==2028==    at 0x4C28F9F: malloc
(vg_replace_malloc.c:236)
==2028==    by 0x40062C: main (test2.c:11)
==2028==
==2028==
==2028== 1 errors in context 3 of 3:
==2028== Conditional jump or move depends on
uninitialised value(s)
==2028==    at 0x40060F: main (test2.c:7)
==2028==
--2028--
```

Fig. 4 Valgrind output of test2.c

III. MEMWATCH

MEMWATCH [2] is a debugging tool for memory error detection and leakage. It is written by Johan Lindh. It can detect memory overflow, underflow, unfreed memory, double frees errors. To use MEMWATCH for memory errors detection add header file to source code and compile. The following program code test3.c (Fig. 5) which includes memory leak, memory overflow, and invalid pointer.

```
#include "stdlib.h"
#include "stdio.h"
#include "memwatch.h"
main()
{
  char *p,*q,*s;
  p=malloc(10);
  q=malloc(10);
  q=p;
  q[10]='a';
  free(p);
  free(q);
  free(s);
}
```

```
sony@sony-
VPCEB14EN:~/Downloads/memwatch-2.71$
gcc -DMEMWATCH -DMW_STDIO test3.c
memwatch.c -o test3op
sony@sony-
VPCEB14EN:~/Downloads/memwatch-2.71$
./test3op
MEMWATCH detected 3 anomalies
```

Fig. 6 compiling test3.c using MEMWATCH

```
NULL free: <5> test3.c(13), NULL pointer free'd
Stopped at Sat Feb  4 21:41:44 2012
unfreed: <2> test3.c(8), 10 bytes at 0x22cd390
          {FE FE FE FE FE FE FE FE FE FE .. .. .. ..
.. .. .....}
Memory usage statistics (global):
 N)umber of allocations made: 2
 L)argest memory usage      : 20
 T)otal of all alloc() calls: 20
 U)nfreed bytes totals      : 10
```

Fig. 7 memwatch.log file

Compile the test3.c and execute the output file as shown in Fig. 6. By observing memwatch.log file we understood how MEMWATCH debugs the program. Fig. 7 show last few lines of memwatch.log file.

The memwatch.log file includes line number at which error occurs and last few lines shows memory usage statistics. The MEMWATCH differs from other debugging tools, here we have debug the program at compile time itself.

IV. strace

strace [4] is debugging utility to monitor the system calls used by the program and all the signals it receives. strace is useful when we do not have the source code and would like to debug the execution of a program. Each line in the strace output contains system call name, followed by its arguments in parentheses and its return value.

Following Fig. 8 shows output of strace for open system call called during mounting usb drive.

```
root@sony-VPCEB14EN:/home/sony# strace -e
trace=open mount -t vfat /dev/sdb1 /mnt
open("/etc/ld.so.cache", O_RDONLY)      = 3
open("/lib/x86_64-linux-gnu/libblkid.so.1",
O_RDONLY) = 3
open("/lib/x86_64-linux-gnu/libselinux.so.1",
O_RDONLY) = 3
open("/lib/x86_64-linux-gnu/libmount.so.1",
O_RDONLY) = 3
open("/lib/x86_64-linux-gnu/libc.so.6",
O_RDONLY) = 3
open("/lib/x86_64-linux-gnu/libuuid.so.1",
O_RDONLY) = 3
open("/lib/x86_64-linux-gnu/libdl.so.2",
O_RDONLY) = 3
open("/proc/filesystems", O_RDONLY)     = 3
open("/usr/lib/locale/locale-archive",
O_RDONLY) = 3
open("/dev/null", O_RDWR)               = 3
open("/etc/mtab", O_RDWR|O_CREAT, 0644) =
3
open("/etc/mtab~.3368",
O_WRONLY|O_CREAT, 0600) = 3
open("/etc/mtab~", O_WRONLY)            = 3
open("/etc/mtab", O_RDONLY)             = 4
open("/etc/mtab.AGAgLo",
O_RDWR|O_CREAT|O_EXCL, 0600) = 4
```

Fig. 8 strace output

The output in fig. 8 shows that during mounting usb drive system called 15 open system call and each returning positive value indicating success.

V. GBD

Along with memory bug, segmentation fault is also major problem during programming. The output of program does not match with requirements this is due to logical mistake or value of variables are not correct. To cope with this situation GDB [5][3] is the debugging solution. GDB, the GNU project debugger is an executable, is the standard debugger for GNU software system. GDB allows us to see what is going on inside another program while it executes or what another program was doing at moment of crash. What can GDB do for debugging?

1. It can start our program, specifying anything that might affect its behavior.
2. It can stop our program on specified condition.
3. We can check what has happened, when our program has stopped.
4. We can change value of variable or condition and check how it could affects after changes.

GDB can run on most UNIX like operating system and Microsoft Windows variants. GDB can debug C, C++, Pascal, many other language programs.

There are many GDB commands, we can use help command to display list of commands. Following is the few lists of commands of GDB with short description:

run- starts program execution from the beginning of the program.
continue- continue execution to next break point.
kill- kill program execution being debug.
quit- exit the GDB debugger.
step- step to next line of code.
next- execute next line of code and will not enter functions.
print- print value of variable or an expression.
break- set breakpoint at specified line or function.
delete- delete all break points or watchpoints.

Now to test the program using GDB consider following program test4.c (Fig. 9). The output of test4.c in Fig. 10 shows segmentation fault.

From output it is clear that after printf statement segmentation fault came. So we apply gdb debugger from line no. 3 that is from char ch declaration using break command and use run command to execute the program. See Fig. 11. After this we have printed value of ch which should be a valid address. But print command showed that 0x0 an invalid address and we were passing this invalid address to printf statement at line no 8. So line no. 3 creating a segmentation fault. To remove this problem we did modification at line no. 3 as char ch[2]=”y” in source code. After this we compiled and test the program using gdb again. Fig. 12 shows output after run command of gdb.

```
#include<stdio.h>
main()
{
  char *ch="y";
  while(!strcmp(ch,"y"))
  {
    printf("Enter character");
    scanf("%s" ,ch);
  }
}
```

Fig. 9 test4.c

```
sony@sony-VPCEB14EN:~/debug$ gcc -g test4.c -o test4op
sony@sony-VPCEB14EN:~/debug$ ./test4op
Enter charactery
Segmentation fault
```

```
sony@sony-VPCEB14EN:~/debug$ gdb ./test4op
…
 (gdb) break 3
Breakpoint 1 at 0x40056c: file test4.c, line 3.
(gdb) run
Starting program: /home/sony/debug/test4op
Breakpoint 1, main () at test4.c:4
4               char *ch="y";
(gdb) print ch
$1 = 0x0
(gdb) quit
A debugging session is active.
Inferior 1 [process 2626] will be killed.
Quit anyway? (y or n) y
```

Fig. 11 gdb output for test4.c

```
 (gdb) run
Starting program: /home/sony/debug/test4op
Enter charactery
Enter charactery
Enter charactery
```

Fig. 12 gdb output after changes in test4.c

Now consider following program test5.c (Fig. 13(a) 13(b) and 13(c)) for catching logical errors using gdb. Fig. 14 shows last few lines of output file

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
 int data;
 struct node *next;
};
int add(int a,struct node **x)
{
 struct node *temp;
 struct node *current=*x;
 if(current==NULL)
 {
  temp=(struct node *)malloc(sizeof(struct
node));
  temp->data=a;
  temp->next=NULL;
  *x=temp;
  return 1;
 }
 else
 {
    if(current->next==NULL)
    {
     temp=(struct node *)malloc(sizeof(struct
node));
     temp->data=a;
     temp->next=NULL;
     current->next=temp;
     return 1;
    }
    else
    {
     while(current->next!=NULL)
     {
      current=current->next;
     }
     temp=(struct node *)malloc(sizeof(struct
node));
     temp->data=a;
```

Fig 13(a) test5.c

test5op of test5.c. From this output it is clear that in delete method somewhere went wrong. Thus we use break command at line no.55 at which delete method is defined and use run command for execution. This is shown in Fig. 15.

```
void display(struct node *p)
 {
  printf("\nlink list:");
  while(p!=NULL)
  {
   printf("%d",p->data);
   p=p->next;
   if(p)
   printf("->");
  }
 }
void delete(struct node *p)
{
 int n;
 struct node *old;
 if(p->next==NULL)
 {
  n=p->data;
  free(p);
  printf("\nNode with data=%d deleted", n);
 }
 else
 {
  while(p->next!=NULL)
  {
   old=p->next;
   p=p->next;
  }
  old->next=NULL;
  n=p->data;
  free(p);
  printf("\nNode with data=%d deleted",n);
 }
}
```

Fig. 13(b) test5.c

Once it reached line number 55 we have used s to step it line by line. We have printed value of old pointer which is pointer of type struct node. During first iteration of while loop the value of data for old pointer should be 1 and value of data for pointer p should be 2, so that old should point to the node previous of node to which p is pointing. But from the output it is clear that both pointers are pointing to same node having data equals to 2. This is logically incorrect. This is due to improper assignments. By simply observing(fig. 13(b)) pointer assignments inside while loop of else part of delete method we

got that we did same assignments for both pointers old and p which is logically incorrect.

```
main()
{
 struct node *p=NULL;
 int  data,n,c=0;
 char ch[2]="y";
 while(!strcmp(ch,"y"))
 {
  printf("\nenter the data");
  scanf("%d",&data);
  n=add(data,&p);
   if(n==1)
   {
    printf("Node with data=%d is added",data);
     c++;
   }
   else
   {
    printf("\nSome error during adding data");
     return;
   }
  printf("\nDo you want to continue y/n:");
  scanf("%s",ch);
 }
 display(p);
 while(c)
 {
  delete(p);
  c--;
  sleep(1);
 }
 printf("\n");
 return 0;
}
```

Fig. 13(c) test5.c

The line number 69 should be changed as old=p to make correct assignment. Fig. 16 shows gdb output of test5.c after proper changes. This time it run properly and also last line showed that program exited normally.

```
sony@sony-VPCEB14EN:~/debug$ gcc -g test5.c -o test5op
sony@sony-VPCEB14EN:~/debug$ ./test5op
6_64-linux-gnu/ld-2.13.so
7fbbc66e7000-7fbbc66e9000 rw-p 00021000 08:07 397305 /lib/x86_64-linux-gnu/ld-2.13.so
7fff610cf000-7fff610f0000 rw-p 00000000 00:00 0 [stack]
7fff611ff000-7fff61200000 r-xp 00000000 00:00 0 [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
Node with data=3 deletedAborted
```

Fig. 14 output of test5.c

```
sony@sony-VPCEB14EN:~/debug$ gdb ./test5op
…
(gdb) break 55
Breakpoint 1 at 0x400836: file test5.c, line 55.
(gdb) run
Starting program: /home/sony/debug/test5op
enter the data1
Node with data=1 is added
Do you want to continue y/n:y
enter the data2
Node with data=2 is added
Do you want to continue y/n:y
enter the data3
Node with data=3 is added
Do you want to continue y/n:n
Breakpoint 1, delete (p=0x602010) at test5.c:59
59          if(p->next==NULL)
(gdb) s
67           while(p->next!=NULL)
(gdb) s
69             old=p->next;
(gdb) s
70             p=p->next;
(gdb) print *old
$1 = {data = 2, next = 0x602050}
```

Fig. 15 gdb output of test5.c

```
sony@sony-VPCEB14EN:~/debug$ gcc -g test5.c -o test5op
sony@sony-VPCEB14EN:~/debug$ gdb ./test5op
GNU gdb (Ubuntu/Linaro 7.3-0ubuntu2) 7.3-2011.08
…
 (gdb) run
Starting program: /home/sony/debug/test5op
enter the data1
Node with data=1 is added
Do you want to continue y/n:y
enter the data2
Node with data=2 is added
Do you want to continue y/n:y
enter the data3
Node with data=3 is added
Do you want to continue y/n:n
link list:1->2->3
Node with data=3 deleted
Node with data=2 deleted
Node with data=1 deleted
[Inferior 1 (process 2899) exited normally]
(gdb)
```

Fig. 16 gdb output after changes in test5.c

VI. CONCLUSION

With the help of debugging tools programmer can save much time to remove bugs and programmer can directly reach to line at which there is an error. The Valgrind and MEMWATCH are very useful for memory management errors such as memory leakage. The strace utility can be used when source code is not available for debugging. By observing system calls listed in strace output programmer can detect cause of the bugs and attempt to capture race conditions. Interactive tools like GDB are very powerful for debugging purpose. GDB's set of commands helps programmer to perform debugging during run time.

VII. REFERENCES

[1] http://valgrind.org/
[2]https://www.ibm.com/developerworks/linux/library/l-debug/
[3]http://www.ibm.com/developerworks/systems/library/es-debug/index.html#listing8
[4] www.linuxjournal.com/
[5] http://www.gnu.org/software/gdb/