课程目标

- 学习使用字符界面gdb来调试Linux 程序
- gdb高级应用
- 图形gdb界面-ddd

反汇编

- objdump -d ./bomb >assemble.txt
  - ./bomb是可执行程序
  - >表示将反汇编结果输出到文件中

- Gdb中，可以使用disassemble命令反汇编
  - Disassemble [参数1] [参数2]
  - 不带参数：默认的反汇编范围是所选择地址附近的汇编代码
  - 单个参数：当然也可以是函数名，因为函数名也是一个地址;
  - 两个参数：就是内存地址范围;

GDB日志输出

- set logging file <文件名>
  - 设置输出的文件名称
- set logging on
  - 输入这个命令后，此后的调试信息将输出到指定文件
- set logging off
  - 输入这个命令，关闭到指定文件的输出

Gdb 简介

- GDB是一个强大的命令行调试工具。大家知道命令行的强大就是在于，其可以形成执行序列，形成脚本。UNIX下的软件全是命令行的，这给程序开发提代供了极大的便利，命令行软件的优势在于，它们可以非常容易的集成在一起，使用几个简单的已有工具的命令，就可以做出一个非常强大的功能。
- GDB相对图形界面的VC++等,命令比较难记,这是命令行界面一大缺点
- Linux 也有基于GDB图形界面的调试器,如gdb insight ,DDD.

GDB功能

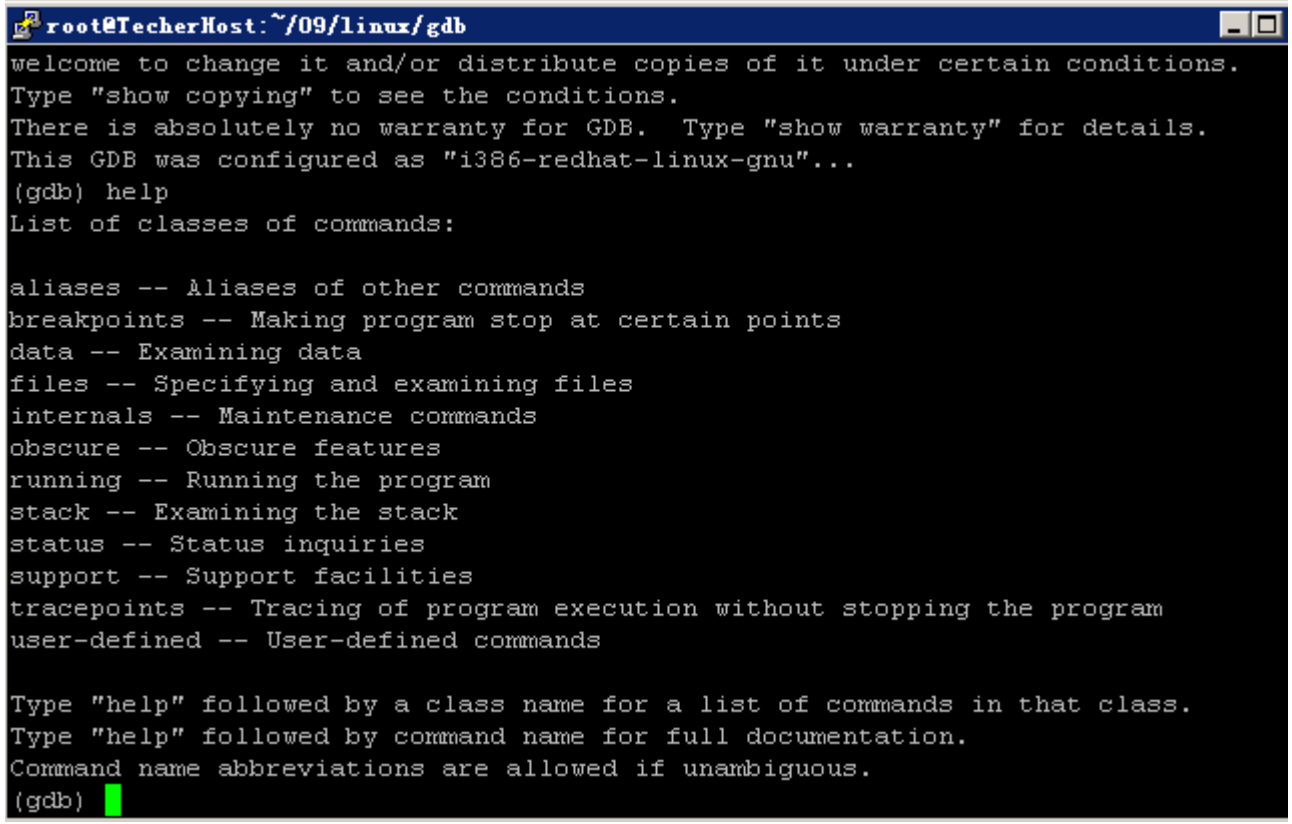
- 启动你的程序，可以按照你的自定义的要求随心所欲的运行程序。
- 可让被调试的程序在你所指定的调置的断点处停住。（断点可以是条件表达式）
- 当程序被停住时，可以检查此时你的程序中所发生的事。
- 动态的改变你程序的执行环境

GDB与VC++调试命令的对比

功能	VC++图标	VC++名称	GDB 命令	GDB 命令缩写
新增，删除断点		Insert/Remove breakpoint	break	**b**
开始调试		go	run	**r**
单步执行，如有函数调用，则进入函数		Step into	step	**s**
单步执行，如有函数调用，则跳过函数		Step over	next	**n**
完成函数调用，并返回		Step out	finish	**f**
运行到光标所在行，GDB 无对应命令，但 continue 类似，		Run to cursor	continue (表示运行到程序结束或下一断点处)	**c**
查看程序调用堆栈		Call Stack	backtrace	**bt**
查看内存		Memory	examine	x
反汇编代码		dissassembly	disassemble	
查看寄存器		Registers	info registers	**i r**
查看变量		Variables	info local	**i lo**
退出调试		Stop debuging	Quit	**q**

GDB帮助

- 象大多部复杂Linux程序一样,GDB是通过内部命令来完成调试工作
- gdb的命令很多，gdb把之分成许多个种类 .
- help命令只是例出gdb的命令种类，如果要看种类中的命令，可以使用help <class> 命令，如：help breakpoints，查看设置断点的所有命令。也可以直接help <command>来查看命令的帮助。
- gdb中，输入命令时，可以不用打全命令，只用打命令的前几个字符就可以了，当然，命令的前几个字符应该要标志着一个唯一的命令
- 在gdb下，你可以敲击两次TAB键来补齐命令的全称，如果有重复的，那么gdb会把其例出来。


root@TecherHost:~/09/linux/gdb
welcome to change it and/or 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.
This GDB was configured as "i386-redhat-linux-gnu"...
(gdb) help
List of classes of commands:
aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands
Type "help" followed by a class name for a list of commands in that class.
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)

GDB测试样例

- 编译测试程序,一定要加上-g参数，为可执行文件加上调试信息
  - gcc -g tst.c -o tst
- 启动GDB的方式
  - gdb <program>
    - program也就是你的执行文件，一般在当前目录下
  - gdb <program> core
    - 用gdb同时调试一个运行程序和core文件，core是程序非法执行后core dump后产生的文件
  - gdb <program> <PID>
    - 如果你的程序是一个服务程序，那么你可以指定这个服务程序运行时的进程ID。gdb会自动attach上去，并调试他。program应该在PATH环境变量中搜索得到。
- 操作GDB常见命令

GDB的启动选项

- GDB启动时，可以加上一些GDB的启动选项
  - --symbols <file>
  - -s <file>
    从指定文件中读取符号表。
  - -se file
    从指定文件中读取符号表信息，并把他用在可执行文件中。
  - --core <file>
    -c <file>
    调试时core dump的core文件。
  - --directory <directory>
    -d <directory>
    加入一个源文件的搜索路径。默认搜索路径是环境变量中PATH所定义的路径。
  - file program
- 详细的开关可以用gdb --help

Linux

培训

gdb的基本调试

调试器的基本功能

- 任何一种调试器，都必须具备如下基本功能
  - 建立可执行程序与源码的联系(查看源码)
  - 设置断点
  - 执行行基本的调试命令
  - 程序暂停后，查看各种信息

查看源程序

- GDB 可以打印出所调试程序的源代码，当然，在程序编译时一定要加上-g的参数，把源程序信息编译到执行文件中。
- 当程序停下来以后，GDB会报告程序停在了那个文件的第几行上。你可以用list命令来打印程序的源代码。可以缩写为l
  - list <linenum>
    显示程序第linenum行的周围的源程序。
  - list <function>
    显示函数名为function的函数的源程序。
  - list
    显示当前行后面的源程序。
  - list -
    显示当前行前面的源程序。
- 一般是打印当前行的上5行和下5行，如果显示函数是是上2行下8行，默认是10行，当然，你也可以定制显示的范围，使用下面命令可以设置一次显示源程序的行数。
  - set listsize <count>
    - 设置一次显示源代码的行数。
    - set listsize 20 设置显示一次20行
  - show listsize
    查看当前listsize的设置。

查看源程序(2)

- list命令还有下面的用法
  - list <first>, <last>
    显示从first行到last行之间的源代码。
  - list , <last>
    显示从当前行到last行之间的源代码。
  - list +
    往后显示源代码。

- 一般来说在list后面可以跟以下这们的参数：
  - <linenum> 行号。
  - <+offset> 当前行号的正偏移量。
  - <-offset> 当前行号的负偏移量。
  - <filename:linenum> 哪个文件的哪一行。
    - **l gdb_tst.c:20**
  - <function> 函数名。
  - <filename:function> 哪个文件中的哪个函数
    - **l gdb_tst.c:main**
  - <*address> 程序运行时的语句在内存中的地址。
    - **l *0x0804835a**

调试程序

- 执行gdb
  - gdb gdb_tst
- 设置断点
  - 通常至少要设一个断点,要不然gdb会直接运行到程序结束.
  - b main #在主函数入口设断点
- 设置命令行参数
  - 如果程序需要用到命令行参数，直接在gdb命令是无法输入
  - set args 可指定运行时参数。
    - 如：gdb>set args 10 20 30 40 50
- 开始调试
  - 进入gdb提示符后,gdb 并没有进调试状态
  - 需要用r,即run进行调试

```
[root@TecherHost gdb]# fg
gdb gdb_tst
q
The program is running.  Exit anyway? (y or n) y
[root@TecherHost gdb]# gdb gdb_tst
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or 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.
This GDB was configured as "i386-redhat-linux-gnu"...
(gdb) b main
Breakpoint 1 at 0x8048382: file gdb_tst.c, line 31.
(gdb) r
Starting program: /home/hxy/E100/gdb/gdb_tst

Breakpoint 1, main () at gdb_tst.c:31
31        long result = 0;
(gdb) p intary
$1 = {1, 16, 12}
(gdb) p *intary@10
$2 = {1, 16, 12, 1, 1, 12, 134513200, 13, 134513764, 4}
(gdb)
```

在GDB中运行程序

- 当以gdb <program>方式启动gdb后，gdb会在PATH路径和当前目录中搜索<program>的源文件。如要确认gdb是否读到源文件，可使用l或list命令，看看gdb是否能列出源代码。
- 在gdb中，运行程序使用r或是run命令。程序的运行，你有可能需要设置下面四方面的事。
  - 程序运行参数。
    - set args 可指定运行时参数。
    - 如：set args 10 20 30 40 50
  - 运行环境
    - path <dir> 可设定程序的运行路径。
    - show paths 查看程序的运行路径。
    - set environment varname [=value] 设置环境变量。如：set env USER=hchen
    - show environment [varname] 查看环境变量。
  - 工作目录。
    - cd <dir> 相当于shell的cd命令。
    - pwd 显示当前的所在目录。
  - 程序的输入输出
    - info terminal 显示你程序用到的终端的模式。
    - 使用重定向控制程序输出。如：run > outfile
    - tty命令可以指写输入输出的终端设备。如：tty /dev/ttyb

常用调试命令

- 当程序被停住了，你可以用continue命令恢复程序的运行直到程序结束，或下一个断点到来。也可以使用step或next命令单步跟踪程序。
  - continue [ignore-count]
    c [ignore-count]
    fg [ignore-count]
    - 恢复程序运行，直到程序结束，或是下一个断点到来。ignore-count表示忽略其后的断点次数。continue，c，fg三个命令都是一样的意思。
  - step <count>
    - 单步跟踪，如果有函数调用，他会进入该函数。进入函数的前提是，此函数被编译有debug信息。很像VC等工具中的step in。后面可以加count也可以不加，不加表示一条条地执行，加表示执行后面的count条指令，然后再停住。
  - next <count>
    - 同样单步跟踪，如果有函数调用，他不会进入该函数。很像VC等工具中的step
  - set step-mode
    set step-mode on
    - 打开step-mode模式，于是，在进行单步跟踪时，程序不会因为没有debug信息而不停住。这个参数有很利于查看机器码。

常用调试命令(2)

- set step-mod off
  - 关闭step-mode模式。
- finish
  - 运行程序，直到当前函数完成返回。并打印函数返回时的堆栈地址和返回值及参数值等信息。相当于VC 的step out
- until 或 u

  当你厌倦了在一个循环体内单步跟踪时，这个命令可以运行程序直到退出循环体。

  until linenum
- stepi 或 si

  nexti 或 ni

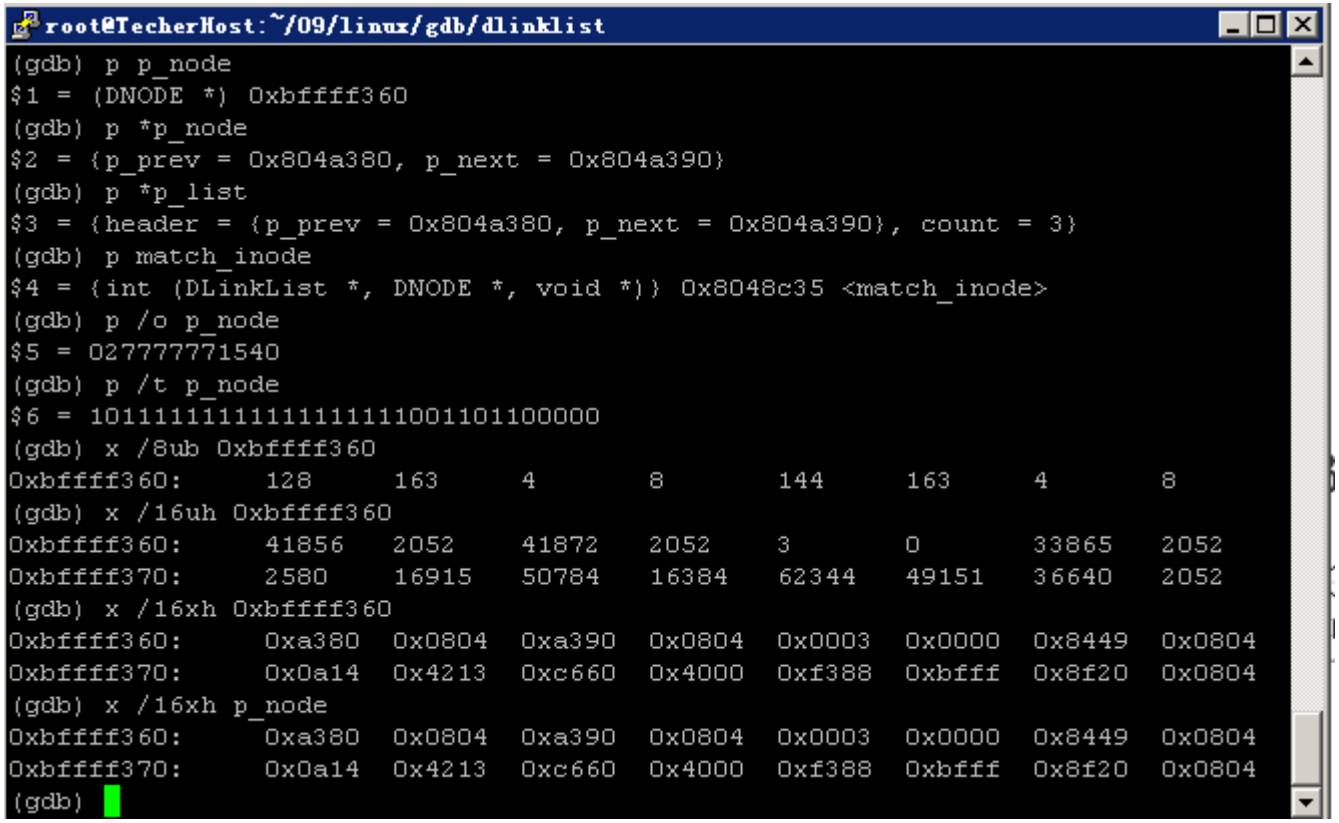
  单步跟踪一条机器指令！一条程序代码有可能由数条机器指令完成，stepi和nexti可以单步执行机器指令。与之一样有相同功能的命令是“display/i $pc”，当运行完这个命令后，单步跟踪会在打出程序代码的同时打出机器指令（也就是汇编代码）

查看运行数据

- GDB主要采用print 来查看运行数据
  - print <expr> #显示表达式值
  - print /f #f表示格式,如 print /x 按十六进制显示
  - printf “x is %d\n”,x #格式化输出
- GDB可以显示被调试程序的const常量、变量、函数 ,但不能显示宏内容
- GDB可以查看三种变量
  - 全局变量（所有文件可见的）
  - 静态全局变量（当前文件可见的）
  - 局部变量（当前Scope可见的）
  - 如果你的局部变量和全局变量发生冲突（也就是重名），一般情况下是局部变量会隐藏全局变量
  - 使用 “::”操作符 强制指定变量所在文件或函数, file::variable , function::variable

查看运行数据(2)

- print 后接结构变量名,则可以把它所有成员打印出来
- 查看数组
  - **p *array@len** # 人为数组，查看内存中的连续对象
  - 如果是静态数组,直接 p array
- print 输出格式
  - x 按十六进制格式显示变量。
  - d 按十进制格式显示变量。
  - u 按十六进制格式显示无符号整型。
  - o 按八进制格式显示变量。
  - t 按二进制格式显示变量。
  - a 按十六进制格式显示变量。**p/a i**
  - c 按字符格式显示变量。**p/c i**
  - f 按浮点数格式显示变量。


root@TecherHost:~/09/linux/gdb/dlinklist
(gdb) p p_node
$1 = (DNODE *) 0xbffff360
(gdb) p *p_node
$2 = {p_prev = 0x804a380, p_next = 0x804a390}
(gdb) p *p_list
$3 = {header = {p_prev = 0x804a380, p_next = 0x804a390}, count = 3}
(gdb) p match_inode
$4 = {int (DLinkList *, DNODE *, void *)} 0x8048c35 <match_inode>
(gdb) p /o p_node
$5 = 027777771540
(gdb) p /t p_node
$6 = 10111111111111111111001101100000
(gdb) x /8ub 0xbffff360
0xbffff360: 128 163 4 8 144 163 4 8
(gdb) x /16uh 0xbffff360
0xbffff360: 41856 2052 41872 2052 3 0 33865 2052
0xbffff370: 2580 16915 50784 16384 62344 49151 36640 2052
(gdb) x /16xh 0xbffff360
0xbffff360: 0xa380 0x0804 0xa390 0x0804 0x0003 0x0000 0x8449 0x0804
0xbffff370: 0x0a14 0x4213 0xc660 0x4000 0xf388 0xbfff 0x8f20 0x0804
(gdb) x /16xh p_node
0xbffff360: 0xa380 0x0804 0xa390 0x0804 0x0003 0x0000 0x8449 0x0804
0xbffff370: 0x0a14 0x4213 0xc660 0x4000 0xf388 0xbfff 0x8f20 0x0804
(gdb)

print显示实例

- (gdb) **p i**
  - $21 = 101
- (gdb) **p/a i**
  - $22 = 0x65
- (gdb) **p/c i**
  - $23 = 101 'e'
- (gdb) **p/f i**
  - $24 = 1.41531145e-43
- (gdb) **p/x i**
  - $25 = 0x65
- (gdb) **p/t i**
  - $26 = 1100101

查看内存

- 使用examine命令（简写是x）来查看内存地址中的值。
- x命令的语法如下所示
  - x/n、f、u是可选的参数
  - n 是一个正整数，表示显示内存的长度，也就是说从当前地址向后显示几个地址的内容。
  - f 表示显示的格式，跟print 的格式参数相同
  - u 表示从当前地址往后请求的字节数，如果不指定的话，GDB默认是4个bytes。u参数可以用下面的字符来代替，b表示单字节，h表示双字节，w表示四字节，g表示八字节。当我们指定了字节长度后，GDB会从指内存定的内存地址开始，读写指定字节，并把其当作一个值取出来。
  - n/f/u三个参数可以一起使用
    - **x/3uh 0x54320** 表示，从内存地址0x54320读取内容，h表示以双字节为一个单位，3表示三个单位，u表示按十六进制显示。

```
23          long result = 0;
(gdb) x /20xb 0x0804835a
0x804835a <main>:       0x55    0x89    0xe5    0x83    0xec    0x08    0x83    0xe4
0x8048362 <main+8>:     0xf0    0xb8    0x00    0x00    0x00    0x00    0x29    0xc4
0x804836a <main+16>:    0xc7    0x45    0xf8    0x00
(gdb) Quit
```

```
31        long result = 0;
(gdb) p intary
$1 = {1, 16, 12}
(gdb) p *intary@10
$2 = {1, 16, 12, 1, 1, 12, 134513200, 13, 134513764, 4}
(gdb) p intary
$3 = {1, 16, 12}
(gdb) p &intary
$4 = (int (*)[3]) 0x80494a4
(gdb) x 0x80494a4
0x80494a4 <intary>:     0x00000001
(gdb) x /3uh 0x80494a4
0x80494a4 <intary>:     1       0       16
(gdb) x /3xh 0x80494a4
0x80494a4 <intary>:     0x0001  0x0000  0x0010
(gdb) x /30xh 0x80494a4
0x80494a4 <intary>:     0x0001  0x0000  0x0010  0x0000  0x000c  0x0000  0x0001 0
x0000
0x80494b4 <_DYNAMIC+4>: 0x0001  0x0000  0x000c  0x0000  0x8230  0x0804  0x000d 0
x0000
0x80494c4 <_DYNAMIC+20>:        0x8464  0x0804  0x0004  0x0000  0x8128  0x0804 0
x0005    0x0000
0x80494d4 <_DYNAMIC+36>:        0x81a0  0x0804  0x0006  0x0000  0x8150  0x0804
(gdb) x /xh 0x80494a4
```

查看栈信息

- 当程序被停住了，你需要做的第一件事就是查看程序是在哪里停住的。当你的程序调用了一个函数，函数的地址，函数参数，函数内的局部变量都会被压入“栈”（Stack）中。你可以用GDB命令来查看当前的栈中的信息。
  - backtrace
    bt
  - backtrace <n>
    bt <n>
    n是一个正整数，表示只打印栈顶上n层的栈信息。
  - backtrace <-n>
    bt <-n>
    -n表一个负整数，表示只打印栈底下n层的栈信息。

```
(gdb) s
func (n=5050) at gdb_tst.c:5
5       int sum=0,i;
(gdb) bt
#0  func (n=5050) at gdb_tst.c:5
#1  0x0804839d in main () at gdb_tst.c:30
#2  0x42015574 in __libc_start_main () from /lib/tls/libc.so.6
(gdb) bt 1
```

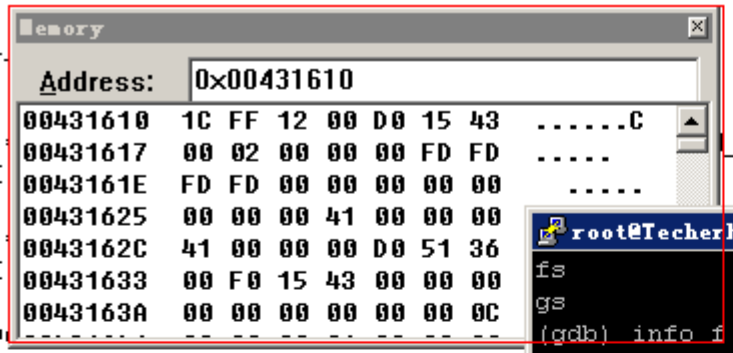
查看栈信息 (2)

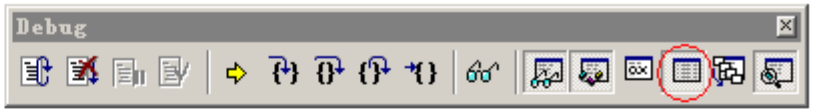
- 如果你要查看某一层的信息，你需要在切换当前的栈，一般来说，程序停止时，最顶层的栈就是当前栈，如果你要查看栈下面层的详细信息，首先要做的是切换当前栈。
  - frame <n>
    f <n>
    - n是一个从0开始的整数，是栈中的层编号。比如：frame 0，表示栈顶，frame 1，表示栈的第二层。
  - up <n>
    - 表示向栈的上面移动n层,并打印栈详细信息.可以不打n,表示向上移动一层。
  - down <n>
    - 表示向栈的下面移动n层，并打印栈详细信息,可以不打n，表示向下移动一层。
- 上面的命令，都会打印出移动到的栈层的信息。如果你不想让其打出信息。你可以使用这三个命令：
  - select-frame <n> 对应于 frame 命令。
  - up-silently <n> 对应于 up 命令。
  - down-silently <n> 对应于 down 命令。
  -

```
(gdb) f 1
#1  0x0804839d in main () at gdb_tst.c:30
30        printf("%d\n",func(result));
(gdb) f  2
#2  0x42015574 in __libc_start_main () from /lib/tls/libc.so.6
(gdb)
```

查看栈信息 (3)

- info frame
  info f
  - 这个命令会打印出更为详细的当前栈层的信息，只不过，大多数都是运行时的内内地址。比如：函数地址，调用函数的地址，被调用函数的地址，目前的函数是由什么样的程序语言写成的、函数参数地址及值、局部变量的地址等等。如：
- info args
  打印出当前函数的参数名及其值。
- info locals
  打印出当前函数中所有局部变量及其值。
- info catch
  打印出当前的函数中的异常处理信息





_HEAD);

wList(p list);

10

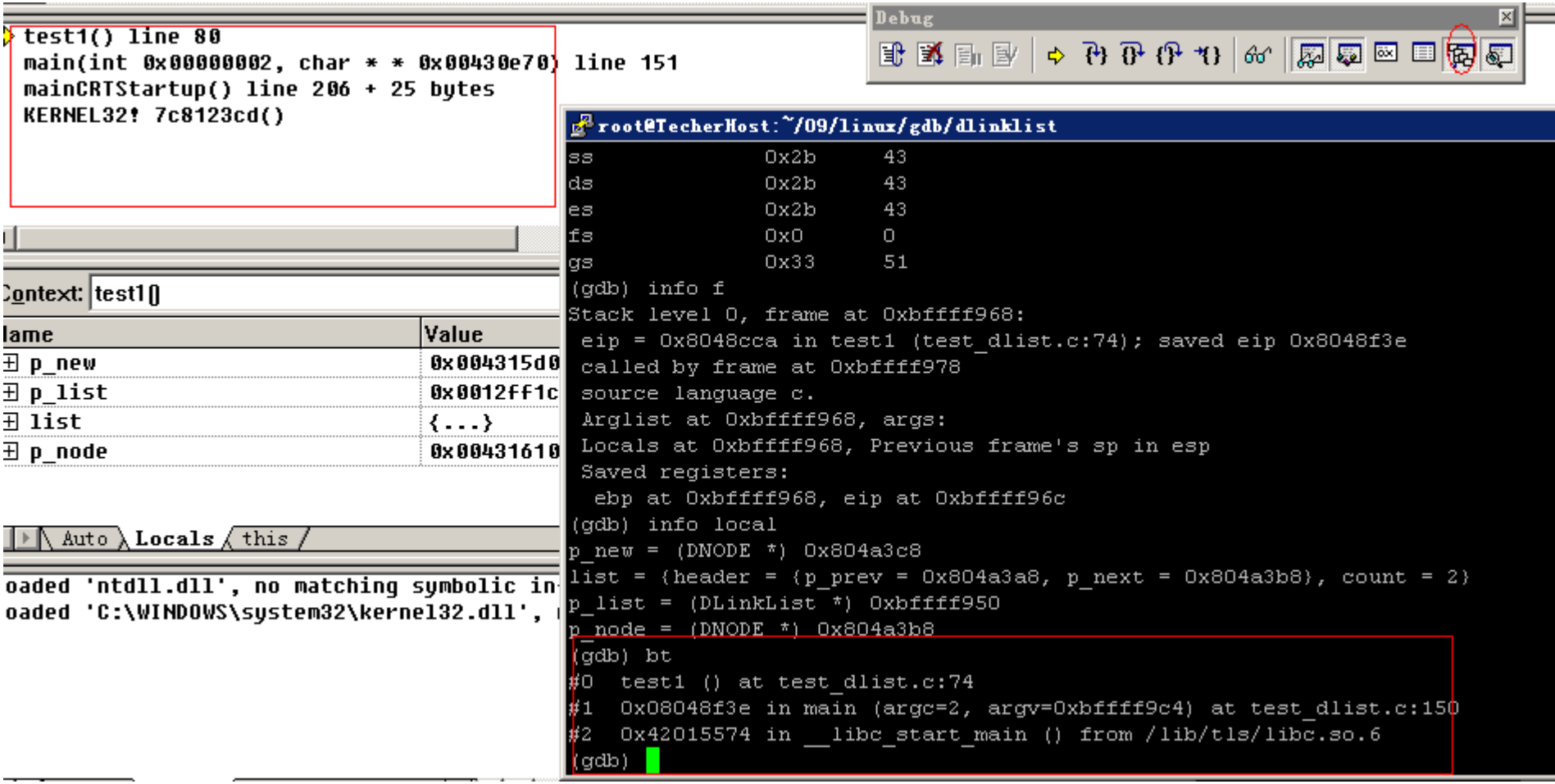
	Value
	0x004315d0
	0x0012ff1c
	{...}
	0x00431610

Locals this

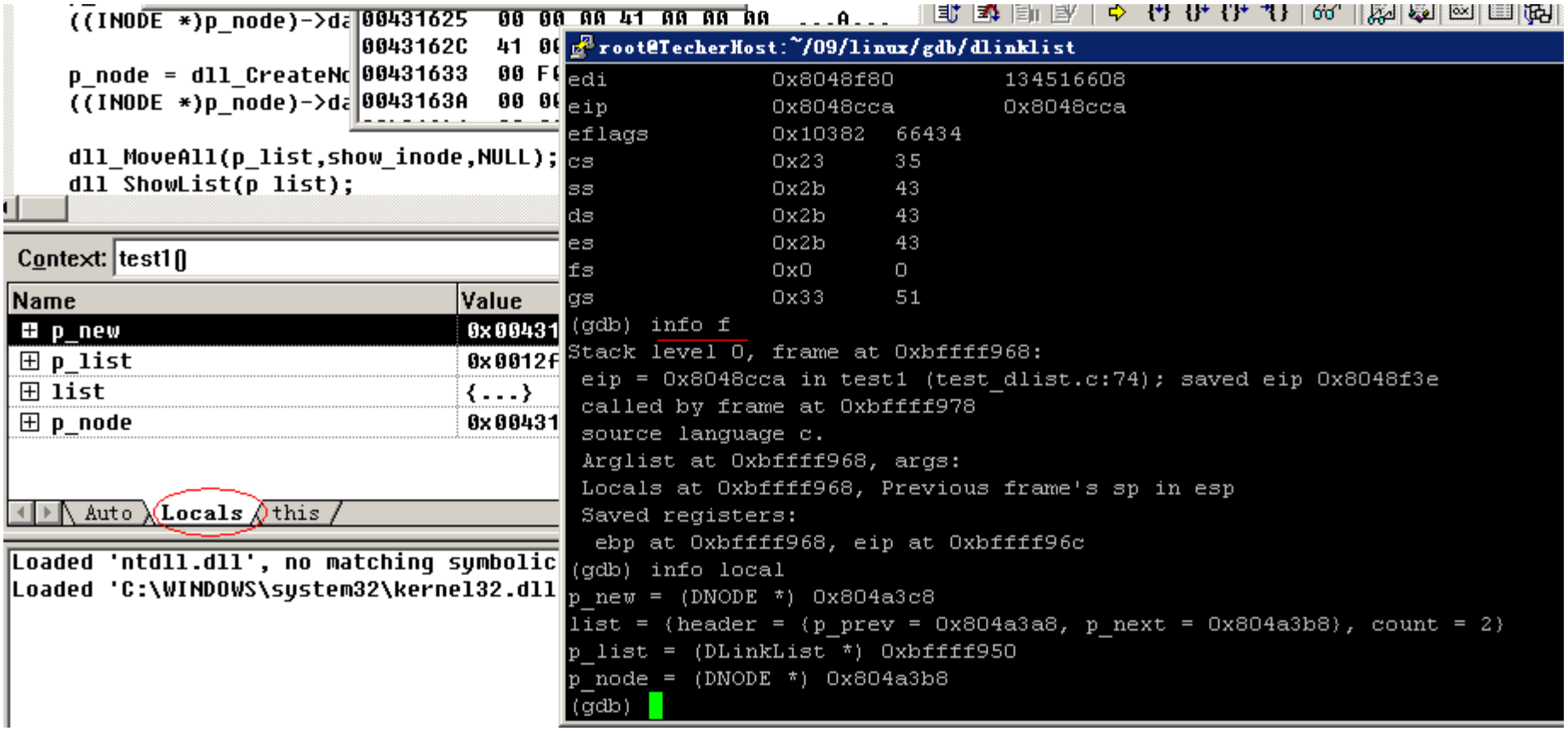
ll.dll', no matching symbolic inf
WINDOWS\system32\kernel32.dll', r

```
root@TecherHost:~/09/linux/gdb/dlinklist
fs              0x0     0
gs              0x33    51
(gdb) info f
Stack level 0, frame at 0xbffff968:
 eip = 0x8048cca in test1 (test_dlist.c:74); saved eip 0x8048f3e
 called by frame at 0xbffff978
 source language c.
 Arglist at 0xbffff968, args:
 Locals at 0xbffff968, Previous frame's sp in esp
 Saved registers:
  ebp at 0xbffff968, eip at 0xbffff96c
(gdb) info local
p_new = (DNODE *) 0x804a3c8
list = {header = {p_prev = 0x804a3a8, p_next = 0x804a3b8}, count = 2}
p_list = (DLinkList *) 0xbffff950
p_node = (DNODE *) 0x804a3b8
(gdb) bt
#0  test1 () at test_dlist.c:74
#1  0x08048f3e in main (argc=2, argv=0xbffff9c4) at test_dlist.c:150
#2  0x42015574 in __libc_start_main () from /lib/tls/libc.so.6
(gdb) x /16uh p_node
0x804a3b8:      63824   49151   41896   2052    2       0       17      0
0x804a3c8:      0       0       0       0       0       0       7217    0
(gdb)
```

查看堆栈的对比


test1() line 80
main(int 0x00000002, char * * 0x00430e70) line 151
mainCRTStartup() line 206 + 25 bytes
KERNEL32! 7c8123cd()
Debug
Context: test1()
Name
Value
p_new
0x004315d0
p_list
0x0012ff1c
list
{...}
p_node
0x00431610
Auto
Locals
this
oaded 'ntdll.dll', no matching symbolic in
oaded 'C:\WINDOWS\system32\kernel32.dll',
root@TecherHost:~/09/linux/gdb/dlinklist
ss 0x2b 43
ds 0x2b 43
es 0x2b 43
fs 0x0 0
gs 0x33 51
(gdb) info f
Stack level 0, frame at 0xbffff968:
eip = 0x8048cca in test1 (test_dlist.c:74); saved eip 0x8048f3e
called by frame at 0xbffff978
source language c.
Arglist at 0xbffff968, args:
Locals at 0xbffff968, Previous frame's sp in esp
Saved registers:
ebp at 0xbffff968, eip at 0xbffff96c
(gdb) info local
p_new = (DNODE *) 0x804a3c8
list = {header = {p_prev = 0x804a3a8, p_next = 0x804a3b8}, count = 2}
p_list = (DLinkList *) 0xbffff950
p_node = (DNODE *) 0x804a3b8
(gdb) bt
#0 test1 () at test_dlist.c:74
#1 0x08048f3e in main (argc=2, argv=0xbffff9c4) at test_dlist.c:150
#2 0x42015574 in __libc_start_main () from /lib/tls/libc.so.6
(gdb)

查看本地变量对比


((INODE *)p_node)->da
p_node = dll_CreateNo
((INODE *)p_node)->da
dll_MoveAll(p_list,show_inode,NULL);
dll_ShowList(p_list);
00431625 00 00 00 41 00 00 00 ...A...
0043162C 41 00
00431633 00 F0
0043163A 00 00
Context: test1()
Name
Value
p_new
0x00431
p_list
0x0012F
list
{...}
p_node
0x00431
Auto
Locals
this
Loaded 'ntdll.dll', no matching symbolic
Loaded 'C:\WINDOWS\system32\kernel32.dll
root@TecherHost:~/09/linux/gdb/dlinklist
edi 0x8048f80 134516608
eip 0x8048cca 0x8048cca
eflags 0x10382 66434
cs 0x23 35
ss 0x2b 43
ds 0x2b 43
es 0x2b 43
fs 0x0 0
gs 0x33 51
(gdb) info f
Stack level 0, frame at 0xbffff968:
eip = 0x8048cca in test1 (test_dlist.c:74); saved eip 0x8048f3e
called by frame at 0xbffff978
source language c.
Arglist at 0xbffff968, args:
Locals at 0xbffff968, Previous frame's sp in esp
Saved registers:
ebp at 0xbffff968, eip at 0xbffff96c
(gdb) info local
p_new = (DNODE *) 0x804a3c8
list = {header = {p_prev = 0x804a3a8, p_next = 0x804a3b8}, count = 2}
p_list = (DLinkList *) 0xbffff950
p_node = (DNODE *) 0x804a3b8
(gdb)

GDB命令示例

- 在进入函数func时，设置一个断点。可以敲入break func，或是直接就是b func
  - b func
- 敲入b按两次TAB键，你会看到所有b打头的命令
  - b
- 只记得函数的前缀，可以输入前缀按tab
  - b make_ <按TAB键>
- 调试C++的程序时，有可以函数名一样,输入前缀按M-?
  - b 'bubble(M-?
- 要退出gdb时，只用发quit或命令简称q就行了

GDB中运行shell命令

- 在gdb环境中，你可以**运行shell命令**，使用gdb的shell命令来完成：
  - shell <command string>
- 调用Linux的shell来执行<command string>，环境变量SHELL中定义的Linux的shell将会被用来执行<command string>，如果SHELL没有定义，那就使用Linux的标准shell：/bin/sh。（在Windows中使用Command.com或cmd.exe）
- make 可直接在gdb执行
  - make <make-args>
  - 可以在gdb中执行make命令来重新build自己的程序。这个命令等价于“shell make <make-args>”。
- Kill

  不退出gdb杀死被调程序，更改程序后重新run保持原有断点信息

Linux

培训

gdb特殊调试方法(选)

暂停 / 恢复程序运行

- 调试程序中，暂停程序运行是必须的，GDB可以方便地暂停程序的运行。你可以设置程序的在哪行停住，在什么条件下停住，在收到什么信号时停往等等。以便于你查看运行时的变量，以及运行时的流程。
- 当进程被gdb停住时，你可以使用info program 来查看程序的是否在运行，进程号，被暂停的原因。
- 在gdb中，我们可以有以下几种暂停方式：断点（BreakPoint）、观察点（WatchPoint）、捕捉点（CatchPoint）、信号（Signals）、线程停止（Thread Stops）。如果要恢复程序运行，可以使用c或是continue命令。
- VC++ 不支持attach调试,也不支持信号和线程停止.

暂停方法(2)设置断点（BreakPoint）

- 我们用break命令来设置断点。有如下设置断点的方法：
  - break <function>
    - 在进入指定函数时停住。C++中可以使用class::function或function(type,type)格式来指定函数名。
  - break <linenum>
    - 在指定行号停住。
  - break +offset
  - break -offset
    - 在当前行号的前面或后面的offset行停住。offiset为自然数。
  - break filename:linenum
    - 在源文件filename的linenum行处停住。
  - break filename:function
    - 在源文件filename的function函数的入口处停住。
  - break *address
    - 在程序运行的内存地址处停住。
  - break
    - break命令没有参数时，表示在下一条指令处停住。
  - break ... if <condition>
  - 可以是上述的参数，condition表示条件，在条件成立时停住。比如在循环境体中，可以设置break if i=100，表示当i为100时停住程序。

暂停方法(3)设置观察点WatchPoint）

- 观察点一般来观察某个表达式（变量也是一种表达式）的值是否有变化了，如果有变化，马上停住程序。我们有下面的几种方法来设置观察点：
  - watch <expr>
    - 为表达式（变量）expr设置一个观察点。一量表达式值有变化时，马上停住程序
  - rwatch <expr>
    - 当表达式（变量）expr被读时，停住程序。
  - awatch <expr>
  - 当表达式（变量）的值被读或被写时，停住程序。
  -

```
(gdb) watch g_test
Hardware watchpoint 1: g_test
(gdb) r
Starting program: /home/hxy/gdb_tst
Hardware watchpoint 1: g_test
Hardware watchpoint 1: g_test
Hardware watchpoint 1: g_test
```

暂停方法(4)设置捕捉点（CatchPoint）

- 你可设置捕捉点来补捉程序运行时的一些事件。如：载入共享库（动态链接库）或是C++的异常。设置捕捉点的格式为：
  - catch <event>
  - 当event发生时,停住程序。event可以是下面的内容：
    - 1、throw 一个C++抛出的异常。（throw为关键字）
    - 2、catch 一个C++捕捉到的异常。（catch为关键字）
    - 3、exec 调用系统调用exec时。（exec为关键字，目前此功能只在HP-UX下有用）
    - 4、fork 调用系统调用fork时。（fork为关键字，目前此功能只在HP-UX下有用）
    - 5、vfork 调用系统调用vfork时。（vfork为关键字，目前此功能只在HP-UX下有用）
    - 6、load 或 load <libname> 载入共享库（动态链接库）时。（load为关键字，目前此功能只在HP-UX下有用）
    - 7、unload 或 unload <libname> 卸载共享库（动态链接库）时。（unload为关键字，目前此功能只在HP-UX下有用）
  - tcatch <event>
    只设置一次捕捉点，当程序停住以后，应点被自动删除。

信号（Signals）

- 信号是一种软中断，是一种处理异步事件的方法。一般来说，操作系统都支持许多信号。尤其是Linux，比较重要应用程序一般都会处理信号。 Linux定义了许多信号，比如SIGINT表示中断字符信号，也就是Ctrl+C的信号，SIGBUS表示硬件故障的信号；SIGCHLD表示子进程状态改变信号；SIGKILL表示终止程序运行的信号，等等。信号量编程是Linux下非常重要的一种技术。
- GDB有能力在你调试程序的时候处理任何一种信号，你可以告诉GDB需要处理哪一种信号。你可以要求GDB收到你所指定的信号时，马上停住正在运行的程序，以供你进行调试。你可以用GDB的handle命令来完成这一功能。
  - handle <signal> <keywords...>
- 在GDB中定义一个信号处理。信号<signal>可以以SIG开头或不以SIG开头，可以用定义一个要处理信号的范围（如：SIGIO-SIGKILL，表示处理从SIGIO信号到SIGKILL的信号，其中包括SIGIO，SIGIOT，SIGKILL三个信号），也可以使用关键字all来标明要处理所有的信号。一旦被调试的程序接收到信号，运行程序马上会被GDB停住，以供调试。其<keywords>可以是以下几种关键字的一个或多个。

信号（Signals）(2)

- Handle 的 keywords
  - nostop
    当被调试的程序收到信号时，GDB不会停住程序的运行，但会打出消息告诉你收到这种信号。
  - stop
    当被调试的程序收到信号时，GDB会停住你的程序。
  - print
    当被调试的程序收到信号时，GDB会显示出一条信息。
  - noprint
    当被调试的程序收到信号时，GDB不会告诉你收到信号的信息。
  - pass
    noignore
    当被调试的程序收到信号时，GDB不处理信号。这表示，GDB会把这个信号交给被调试程序会处理。
  - nopass
    ignore
    当被调试的程序收到信号时，GDB不会让被调试程序来处理这个信号。

线程（Thread Stops）

- 如果你程序是多线程的话，你可以定义你的断点是否在所有的线程上，或是在某个特定的线程。GDB很容易帮你完成这一工作。
  - break <linespec> thread <threadno>
  - break <linespec> thread <threadno> if ...
  - linespec指定了断点设置在的源程序的行号。threadno指定了线程的ID，注意，这个ID是GDB分配的，你可以通过“info threads”命令来查看正在运行程序中的线程信息。如果你不指定thread <threadno>则表示你的断点设在所有线程上面。你还可以为某线程指定断点条件。
    - (gdb) break frik.c:13 thread 28 if bartab > lim
  - 当你的程序被GDB停住时，所有的运行线程都会被停住。这方便你你查看运行程序的总体情况。而在你恢复程序运行时，所有的线程也会被恢复运行。那怕是主进程在被单步调试时。

查看信息 info

- 查看断点时，（注：n表示断点号）
  - info breakpoints [n]
  - info break [n]
- 列出当前所设置了的所有观察点。
  - info watchpoints
- 查看有哪些信号在被GDB检测中
  - info signals
    info handle

维护停止点

- 上面说了如何设置程序的停止点，GDB中的停止点也就是上述的三类。在GDB中，如果你觉得已定义好的停止点没有用了，你可以使用delete、clear、disable、enable这几个命令来进行维护。
  - clear <function>
  - clear <filename:function>
    - 清除所有设置在函数上的停止点。
  - delete [breakpoints] [range...]
    - 删除指定的断点，breakpoints为断点号。如果不指定断点号，则表示删除所有的断点。range表示断点号的范围（如：3-7）。其简写命令为d。

维护停止点(2)

- 比删除更好的一种方法是disable停止点，disable了的停止点，GDB不会删除，当你还需要时，enable即可，就好像回收站一样。
  - disable [breakpoints] [range...]
    - disable所指定的停止点，breakpoints为停止点号。如果什么都不指定，表示disable所有的停止点。简写命令是dis.
  - enable [breakpoints] [range...]
    - enable所指定的停止点,breakpoints为停止点号。
  - enable [breakpoints] once range...
    - enable所指定的停止点一次，当程序停止后，该停止点马上被GDB自动disable。
  - enable [breakpoints] delete range...
    - enable所指定的停止点一次，当程序停止后，该停止点马上被GDB自动删除。

停止条件维护

- 前面在说到设置断点时，我们提到过可以设置一个条件，当条件成立时，程序自动停止，这是一个非常强大的功能.
- 条件有相关维护命令。一般来说，为断点设置一个条件，我们使用if关键词，后面跟其断点条件。并且，条件设置好后，我们可以用condition命令来修改断点的条件。（只有break和watch命令支持if，catch目前暂不支持if）
  - condition <bnum> <expression>
    - 修改断点号为bnum的停止条件为expression。
  - condition <bnum>
    - 清除断点号为bnum的停止条件。
- 还有一个比较特殊的维护命令ignore，你可以指定程序运行时，忽略停止条件几次。
  - ignore <bnum> <count>

    表示忽略断点号为bnum的停止条件count次。

为停止点设定运行命令

- 我们可以使用GDB提供的command命令来设置停止点的运行命令。也就是说，当运行的程序在被停止住时，我们可以让其自动运行一些别的命令，这很有利行自动化调试。对基于GDB的自动化调试是一个强大的支持。
  - commands [bnum]
    ... command-list ...
    end
  - 为断点号bnum指写一个命令列表。当程序被该断点停住时，gdb会依次运行命令列表中的命令。
- 如果你要清除断点上的命令序列，那么只要简单的执行一下commands命令，并直接在打个end就行了。

断点菜单

- 在C++中，可能会重复出现同一个名字的函数若干次（函数重载），在这种情况下，break <function>不能告诉GDB要停在哪个函数的入口。当然，你可以使用break <function(type)>也就是把函数的参数类型告诉GDB，以指定一个函数。否则的话，GDB会给你列出一个断点菜单供你选择你所需要的断点。你只要输入你菜单列表中的编号就可以了.
- GDB列出了所有的重载函数，你可以选一下列表编号就行了。0表示放弃设置断点，1表示所有函数都设置断点。

搜索源代码

- 向前面搜索。
  - forward-search <regexp>
  - search <regexp>
- 全部搜索。
  - reverse-search <regexp>
- 其中，<regexp>就是正则表达式，也主一个字符串的匹配模式
-

源代码处理

- 指定源文件的路径
  - 某些时候，用-g编译过后的执行程序中只是包括了源文件的名字，没有路径名。GDB提供了可以让你指定源文件的路径的命令，以便GDB进行搜索。
  - directory <dirname ... >
    dir <dirname ... >
    加一个源文件路径到当前路径的前面。如果你要指定多个路径，UNIX下你可以使用“:”，Windows下你可以使用“;”。
  - directory
    清除所有的自定义的源文件搜索路径信息。
  - show directories
    显示定义了的源文件搜索路径。
- 查看源代码的内存
  - 你可以使用info line命令来查看源代码在内存中的地址。info line后面可以跟“行号”，“函数名”，“文件名:行号”，“文件名:函数名”，这个命令会打印出所指定的源码在运行时的内存地址，
- 查看源码的汇编
  - disassemble func

GDB环境变量

- 你可以在GDB的调试环境中定义自己的变量，用来保存一些调试程序中的运行数据。要定义一个GDB的变量很简单只需。使用GDB的set命令。GDB的环境变量和UNIX一样，也是以$起头。
  - set $foo = *object_ptr
- 使用环境变量时，GDB会在你第一次使用时创建这个变量，而在以后的使用中，则直接对其赋值。环境变量没有类型，你可以给环境变量定义任一的类型。包括结构体和数组。
  - show convenience

    该命令查看当前所设置的所有的环境变量。
-

改变程序的执行

- 一旦使用GDB挂上被调试程序，当程序运行起来后，你可以根据自己的调试思路来动态地在GDB中更改当前被调试程序的运行线路或是其变量的值，这个强大的功能能够让你更好的调试你的程序，
- **修改变量值**
  - 修改被调试程序运行时的变量值，在GDB中很容易实现，使用GDB的print命令即可完成。如：
    - print x=4
- **跳转执行**
  - 一般来说，被调试程序会按照程序代码的运行顺序依次执行。GDB提供了乱序执行的功能，也就是说，GDB可以修改程序的执行顺序，可以让程序执行随意跳跃。这个功能可以由GDB的jump命令来完成：
  - jump <linespec>
    - 指定下一条语句的运行点。<linespce>可以是文件的行号，可以是file:line格式，可以是+num这种偏移量格式。表式着下一条运行语句从哪里开始。
  - jump <address>
    - 这里的<address>是代码行的内存地址。
  - 注意，jump命令不会改变当前的程序栈中的内容，所以，当你从一个函数跳到另一个函数时，当函数运行完返回时进行弹栈操作时必然会发生错误，可能结果还是非常奇怪的，甚至于产生程序Core Dump。所以最好是同一个函数中进行跳转。

改变程序的执行(2)

- 产生信号量
  - 使用singal命令，可以产生一个信号量给被调试的程序。如：中断信号Ctrl+C。这非常方便于程序的调试，可以在程序运行的任意位置设置断点，并在该断点用GDB产生一个信号量，这种精确地在某处产生信号非常有利程序的调试。
  - signal <singal>
    - Linux的系统信号量通常从1到15。所以<singal>取值也在这个范围。
- 强制函数返回
  - 如果你的调试断点在某个函数中，并还有语句没有执行完。你可以使用return命令强制函数忽略还没有执行的语句并返回。
  - return
    return <expression>
    - 使用return命令取消当前函数的执行，并立即返回，如果指定了<expression>，那么该表达式的值会被认作函数的返回值。
- 强制调用函数
  - call <expr>
    p func
  - 表达式中可以一是函数，以此达到强制调用函数的目的。并显示函数的返回值，如果函数返回值是void，那么就不显示。
  - 另一个相似的命令也可以完成这一功能——print，print后面可以跟表达式，所以也可以用他来调用函数，print和call的不同是，如果函数返回void，call则不显示，print则显示函数返回值，并把该值存入历史数据中。

Linux

培训

gdb特殊应用

调试已运行的程序

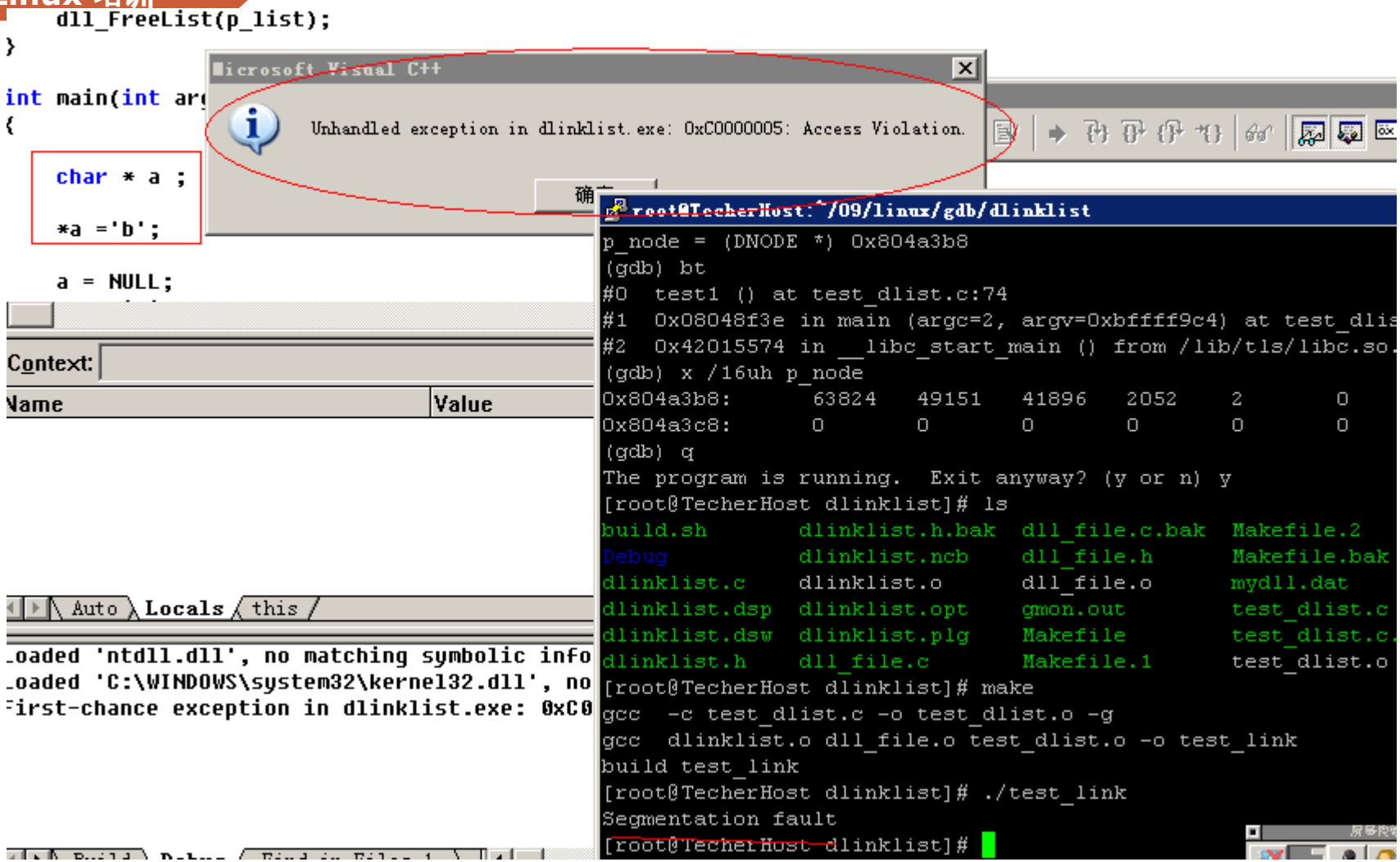
- 两种方法
  - 在Linux下用ps查看正在运行的程序的PID（进程ID），然后用gdb <program> PID格式挂接正在运行的程序。
  - 先用gdb <program>关联上源代码，并进行gdb，在gdb中用attach命令来挂接进程的PID。并用detach来取消挂接的进程。
- 这是调试守护进程常用这两种方法

调试已运行进程

- 运行程序,并用ps查看进程编号
  - ./server &
- 进入gdb
  - gdb server
- 设置断点
- 挂接进程
  - attach <pid>挂接进程，如attach 2595
  - 此时被挂接进程会暂停下来
- 用continue命令恢复运行，进行调试
  - C
- 正常调试阶段

使用gdb查找段错误

- 在Linux下,程序经常出现段错误(segment fault)
  - 通常是访问错误的地址,如向空指针赋值,访问不存在的地址,写到不可能区段
- 产生段错误原因
  - 1)访问系统数据区，尤其是往 系统保护的内存地址写数据,向空地址写值也归属此类
  - 2)内存越界(数组越界，变量类型不一致等) 访问到不属于你的内存区域
- 在Linux下,出现段错误会触发SIGSEGV信号
  - 这个信号的缺省结果打印段错误,并产生一个core dump文件
  - 用gdb打开core dump文件会快速定位出现段错误代码


dll_FreeList(p_list);
}
int main(int ar
{
char * a ;
*a ='b';
a = NULL;
Microsoft Visual C++
Unhandled exception in dlinklist.exe: 0xC0000005: Access Violation.
确
Context:
Name
Value
Auto
Locals
this
Loaded 'ntdll.dll', no matching symbolic info
Loaded 'C:\WINDOWS\system32\kernel32.dll', no
First-chance exception in dlinklist.exe: 0xC0
root@TecherHost: /09/linux/gdb/dlinklist
p_node = (DNODE *) 0x804a3b8
(gdb) bt
#0 test1 () at test_dlist.c:74
#1 0x08048f3e in main (argc=2, argv=0xbffff9c4) at test_dlis
#2 0x42015574 in __libc_start_main () from /lib/tls/libc.so.
(gdb) x /16uh p_node
0x804a3b8: 63824 49151 41896 2052 2 0
0x804a3c8: 0 0 0 0 0 0
(gdb) q
The program is running. Exit anyway? (y or n) y
[root@TecherHost dlinklist]# ls
build.sh dlinklist.h.bak dll_file.c.bak Makefile.2
Debug dlinklist.ncb dll_file.h Makefile.bak
dlinklist.c dlinklist.o dll_file.o mydll.dat
dlinklist.dsp dlinklist.opt gmon.out test_dlist.c
dlinklist.dsw dlinklist.plg Makefile test_dlist.c.
dlinklist.h dll_file.c Makefile.1 test_dlist.o
[root@TecherHost dlinklist]# make
gcc -c test_dlist.c -o test_dlist.o -g
gcc dlinklist.o dll_file.o test_dlist.o -o test_link
build test_link
[root@TecherHost dlinklist]# ./test_link
Segmentation fault
[root@TecherHost dlinklist]#

段错误实例

```
#include <stdio.h>

int main()
{
  char * a = 0x00;

  *a = 100;

  return 0;
}
```

Seg_err.c

生成Core Dump文件

- 缺省Linux不生成Core Dump
  - 用ulimit –c xxx来改变尺寸,生成Core Dump文件
- 当产生段错误时,在可执行文件同一目录生成core.XXX的文件

```
[root@TecherHost gdb]# ls
seg_err.c  seg_err.c.bak
[root@TecherHost gdb]# gcc -g -rdynamic seg_err.c
seg_err.c:12:3: warning: no newline at end of file
[root@TecherHost gdb]# ls
a.out  seg_err.c  seg_err.c.bak
[root@TecherHost gdb]# ./a.out
Segmentation fault
[root@TecherHost gdb]# ls
a.out  seg_err.c  seg_err.c.bak
[root@TecherHost gdb]# ulimit -c 1000
[root@TecherHost gdb]# ./a.out
Segmentation fault (core dumped)
[root@TecherHost gdb]# ls
a.out  core.29478  seg_err.c  seg_err.c.bak
[root@TecherHost gdb]#
```

只提示段错误

改变Core Dump限制

生成Dump文件

用gdb 打开Core dump定位

- gdb <可执行文件> <core文件名>

```
[root@TecherHost gdb]# gdb a.out core.29478
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or 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.
This GDB was configured as "i386-redhat-linux-gnu"...
Core was generated by `./a.out'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/tls/libc.so.6...done.
Loaded symbols for /lib/tls/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x080485c4 in dummy_function () at seg_err.c:4
4       *ptr = 0x00;
(gdb)
```

Linux

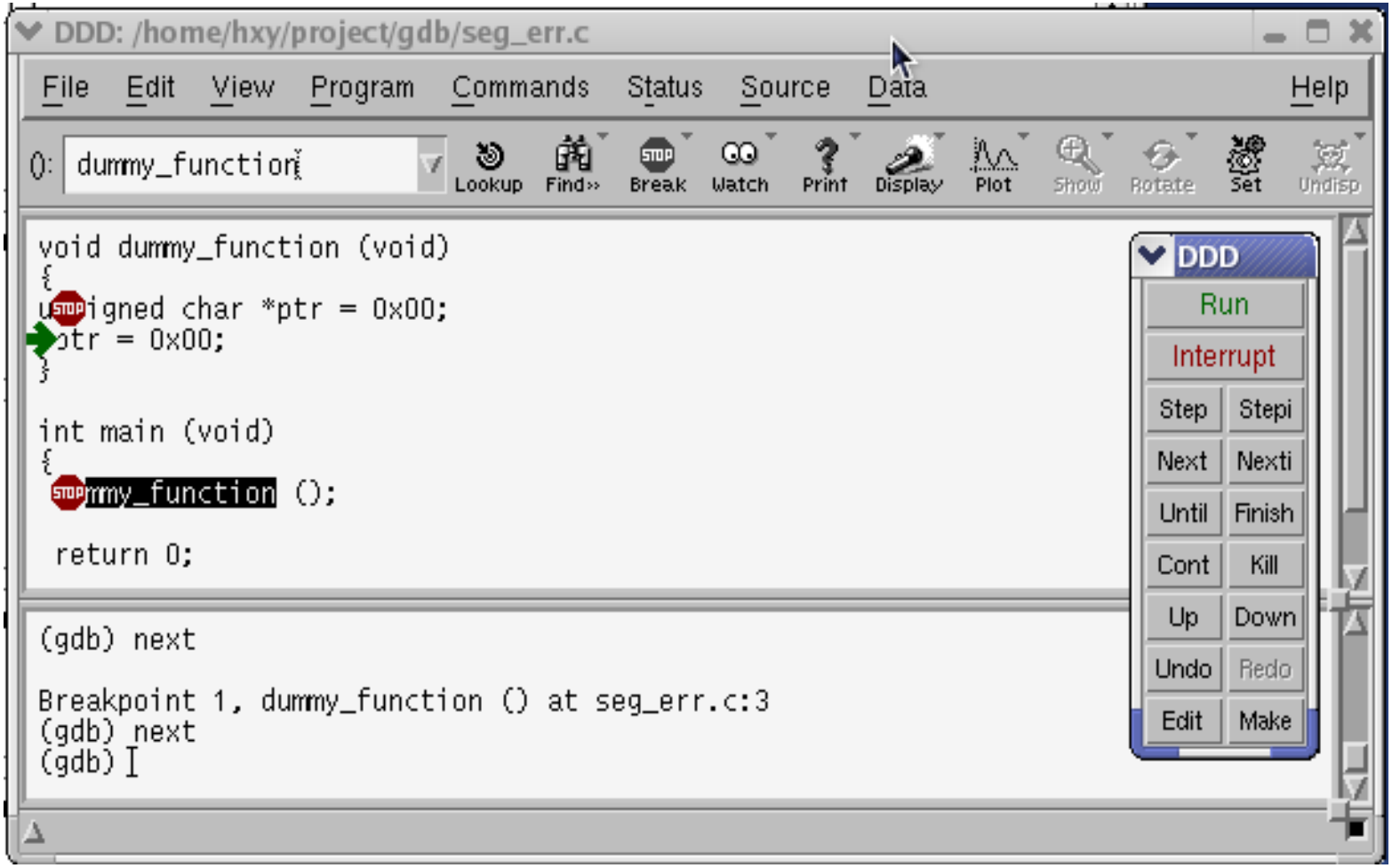
培训

ddd-gdb的图形版

ddd的编译安装

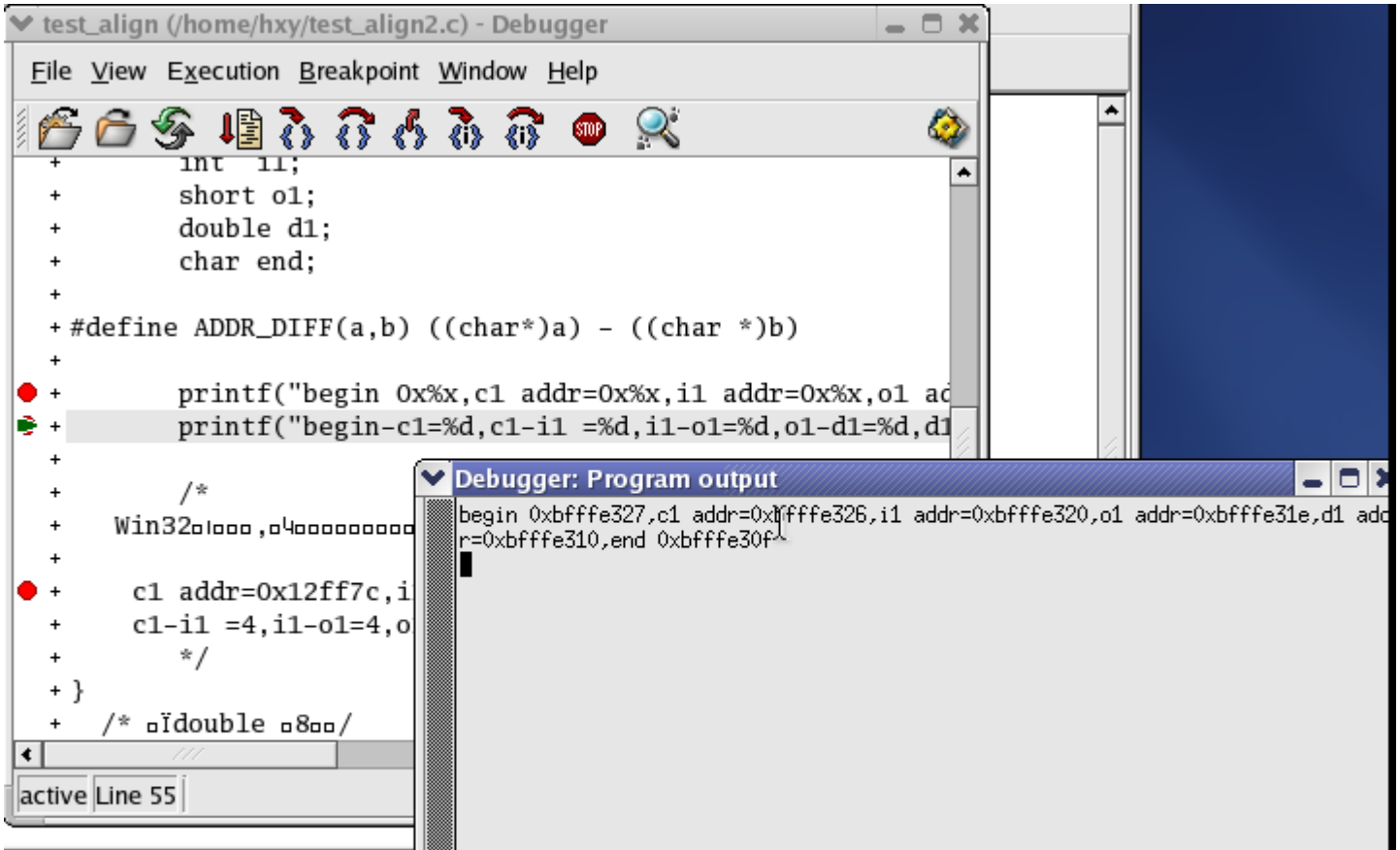
- 要求安装gdb和图形界面
- 安装步骤
  1. 解压 :tar xvzf ddd-3.3.10.tar.gz
  2. 生成Makefile: cd ddd-3.3.10;./configure
  3. 编译 make
  4. 安装 make install
- 运行ddd
  - 直接在命令行下输入ddd即可
  - 或输入调试程序名字 ddd gdb_tst

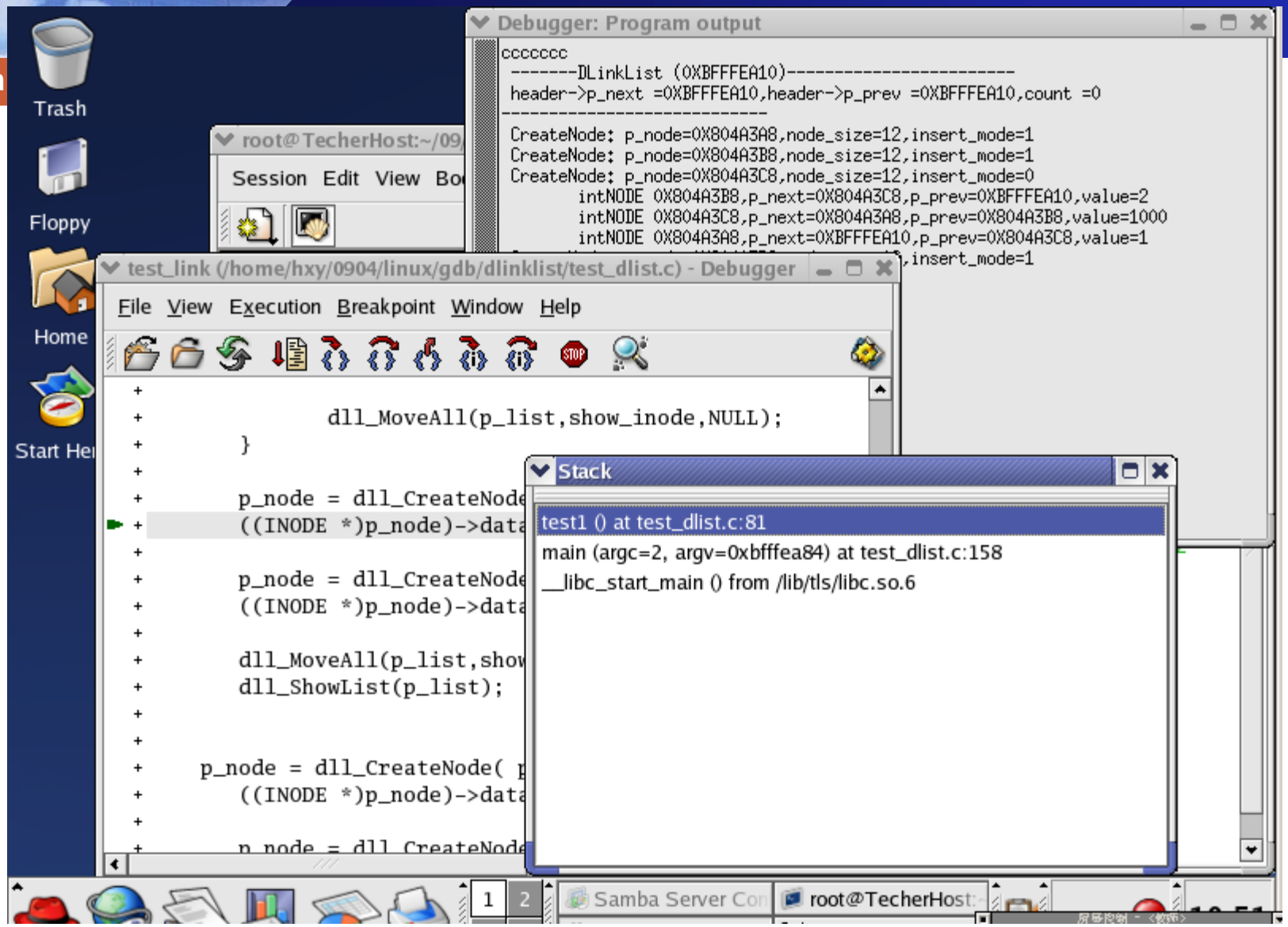
ddd运行效果

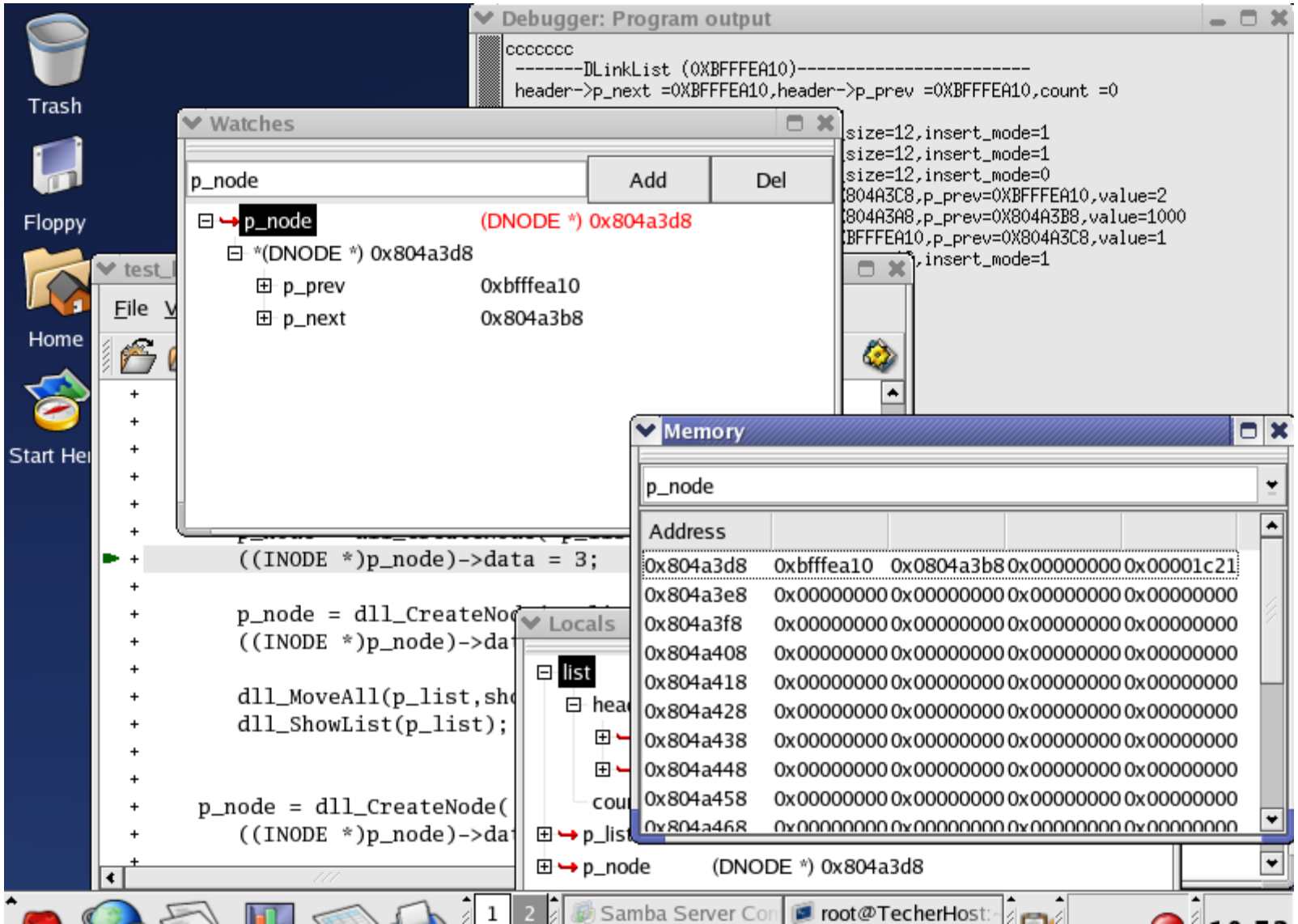

DDD: /home/hxy/project/gdb/seg_err.c
File Edit View Program Commands Status Source Data Help
dummy_function
Lookup Find» Break Watch Print Display Plot Show Rotate Set Undisp
void dummy_function (void)
{
unsigned char *ptr = 0x00;
*ptr = 0x00;
}
int main (void)
{
dummy_function ();
return 0;
(gdb) next
Breakpoint 1, dummy_function () at seg_err.c:3
(gdb) next
(gdb)
DDD
Run
Interrupt
Step Stepi
Next Nexti
Until Finish
Cont Kill
Up Down
Undo Redo
Edit Make

Debugger进行调试

- KDE自带Debugger也是一个图形调试器
  - 调试时用File->Executable...打开一个带调试信息的可执行文件，它将自动打开同一目录下的源码




Trash
Floppy
Home
Start Her
Debugger: Program output
ccccccc
-------DLinkList (0XBFFFEA10)-----------------------
header->p_next =0XBFFFEA10,header->p_prev =0XBFFFEA10,count =0
CreateNode: p_node=0X804A3A8,node_size=12,insert_mode=1
CreateNode: p_node=0X804A3B8,node_size=12,insert_mode=1
CreateNode: p_node=0X804A3C8,node_size=12,insert_mode=0
intNODE 0X804A3B8,p_next=0X804A3C8,p_prev=0XBFFFEA10,value=2
intNODE 0X804A3C8,p_next=0X804A3A8,p_prev=0X804A3B8,value=1000
intNODE 0X804A3A8,p_next=0XBFFFEA10,p_prev=0X804A3C8,value=1
root@TecherHost:~/09
Session Edit View Bo
test_link (/home/hxy/0904/linux/gdb/dlinklist/test_dlist.c) - Debugger
File View Execution Breakpoint Window Help
dll_MoveAll(p_list,show_inode,NULL);
}
p_node = dll_CreateNode
((INODE *)p_node)->data
p_node = dll_CreateNode
((INODE *)p_node)->data
dll_MoveAll(p_list,show
dll_ShowList(p_list);
p_node = dll_CreateNode(p
((INODE *)p_node)->data
Stack
test1 () at test_dlist.c:81
main (argc=2, argv=0xbfffea84) at test_dlist.c:158
__libc_start_main () from /lib/tls/libc.so.6
1
2
Samba Server Con
root@TecherHost:


Trash
Floppy
Home
Start Her
Debugger: Program output
ccccccc
-------DLinkList (0XBFFFEA10)----------
header->p_next =0XBFFFEA10,header->p_prev =0XBFFFEA10,count =0
size=12,insert_mode=1
size=12,insert_mode=1
size=12,insert_mode=0
804A3C8,p_prev=0XBFFFEA10,value=2
804A3A8,p_prev=0X804A3B8,value=1000
BFFFEA10,p_prev=0X804A3C8,value=1
,insert_mode=1
Watches
p_node
Add
Del
p_node (DNODE *) 0x804a3d8
*(DNODE *) 0x804a3d8
p_prev 0xbfffea10
p_next 0x804a3b8
test_l
File V
((INODE *)p_node)->data = 3;
p_node = dll_CreateNo
((INODE *)p_node)->da
dll_MoveAll(p_list,sh
dll_ShowList(p_list);
p_node = dll_CreateNode(
((INODE *)p_node)->da
Memory
p_node
Address
0x804a3d8 0xbfffea10 0x0804a3b8 0x00000000 0x00001c21
0x804a3e8 0x00000000 0x00000000 0x00000000 0x00000000
0x804a3f8 0x00000000 0x00000000 0x00000000 0x00000000
0x804a408 0x00000000 0x00000000 0x00000000 0x00000000
0x804a418 0x00000000 0x00000000 0x00000000 0x00000000
0x804a428 0x00000000 0x00000000 0x00000000 0x00000000
0x804a438 0x00000000 0x00000000 0x00000000 0x00000000
0x804a448 0x00000000 0x00000000 0x00000000 0x00000000
0x804a458 0x00000000 0x00000000 0x00000000 0x00000000
0x804a468 0x00000000 0x00000000 0x00000000 0x00000000
Locals
list
p_node (DNODE *) 0x804a3d8
1
2
Samba Server Con
root@TecherHost:

课堂练习

- 设计一个字符串比较程序，把一个数组中多个字符串进行排序．并按降序输出，
  - 用GDB进行调试
- 用gdb调试有问题的程序

Linux
培训

谢谢，请提问

在疯狂的时代把握未来