Introduction

A quick primer for those who prefer to use a command line debugger. Both gbd (Linux) and dbx (Unix) are very similar. In addition, NuMega technologies produces a very powerful Ring 0 command line debugger for Micro$oft Windows named SoftICE (retail $1000). SoftICE has the same 'feel' as gbd and dbx. Skills learned in one should quickly port to the others.

The Program

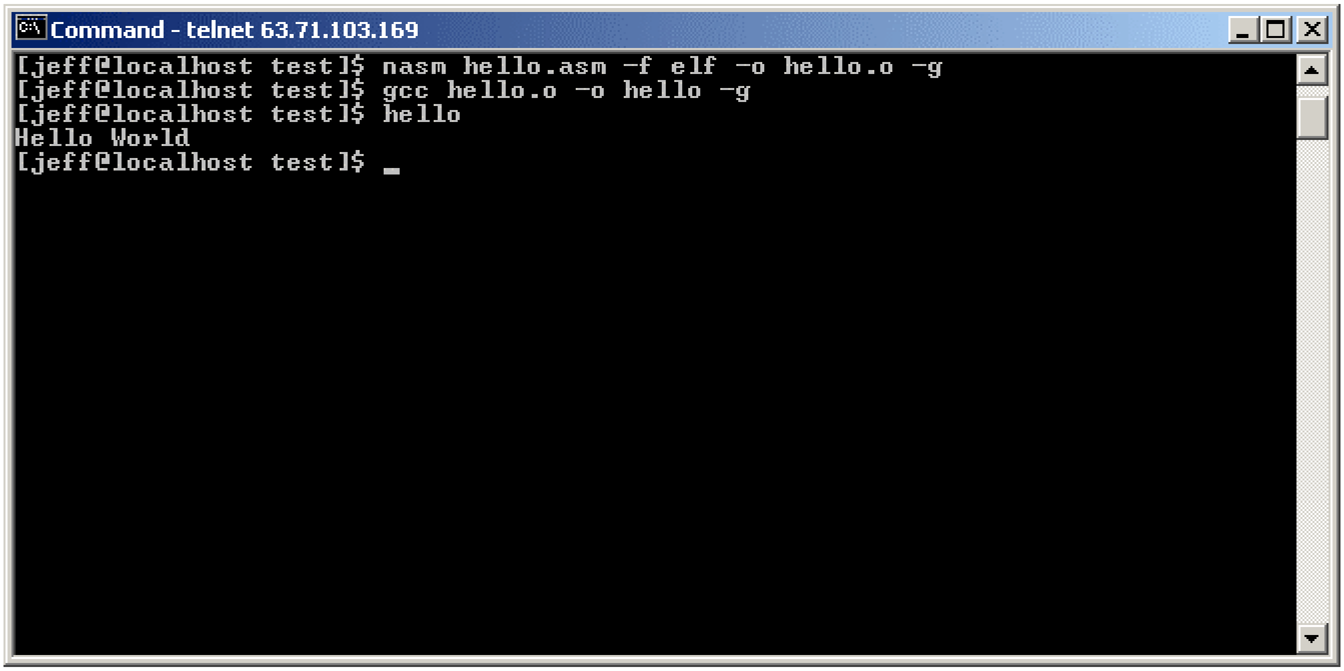
The program used in this document is listed below. It prints the familiar "Hello World" from the nice folks at Bell Labs. In addition, we'll snoop around while the target is under gdb to see if we can find any goodies.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  compile:
;;
;;  nasm hello.asm -f elf -o hello.o -g
;;
;;    -f: elf file format
;;    -o: output file name
;;    -g: debugging information
;;
;;  gcc hello.o -o hello -g
;;
;;    -o: output file name
;;    -g: debugging information
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

GLOBAL main

EXTERN printf

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; defines
;;

LF          equ   0xA       ;; 10 decimal
CR          equ   0xD       ;; 13 decimal
TERM        equ     0       ;; NULL

SYSTEM_EXIT equ     1       ;; exit to OS
SYSTEM_SVC  equ  0x80       ;; int 80h

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; global read/write data
;;

SECTION .data
szHello     db 'Hello World', LF, CR, TERM
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; code
;;

SECTION .text

  main:

    push dword szHello      ;; push address of szHello
    call printf             ;; call c runtime
    add  esp, 4             ;; adjust stack

    mov  eax, SYSTEM_EXIT   ;; prepare for exit
    int  SYSTEM_SVC         ;; good bye

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Compile the Program

Compile and run the program as shown below.

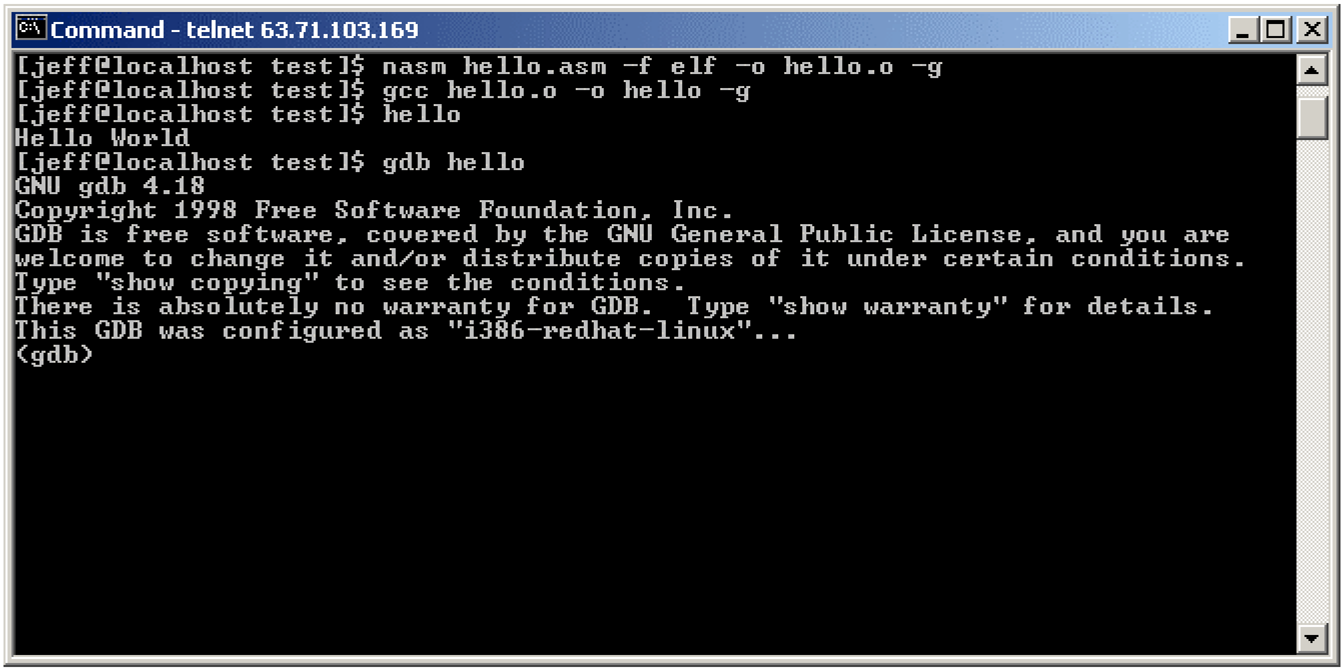


```
[jeff@localhost test]$ nasm hello.asm -f elf -o hello.o -g
[jeff@localhost test]$ gcc hello.o -o hello -g
[jeff@localhost test]$ hello
Hello World
[jeff@localhost test]$ _
```

Works as expected.

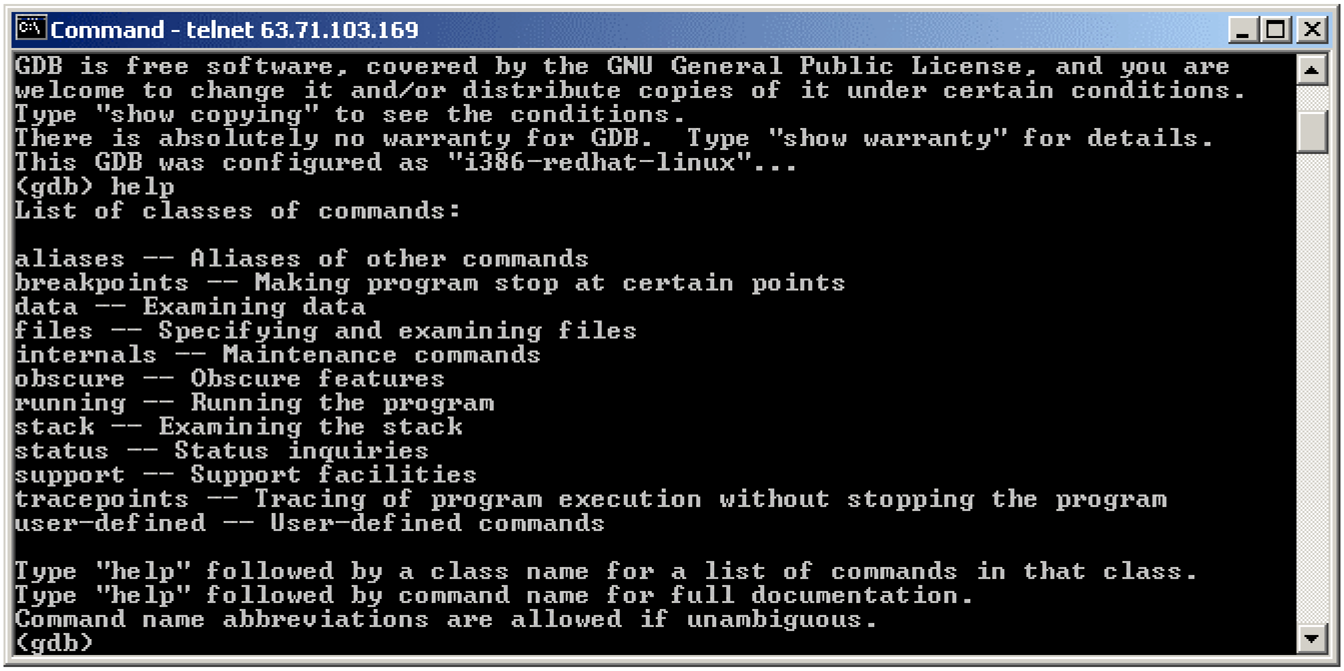
Lets see what's going on under the hood…

Fire up gdb. Execute 'gdb hello' as shown below:



```
[jeff@localhost test]$ nasm hello.asm -f elf -o hello.o -g
[jeff@localhost test]$ gcc hello.o -o hello -g
[jeff@localhost test]$ hello
Hello World
[jeff@localhost test]$ gdb hello
GNU gdb 4.18
Copyright 1998 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"...
(gdb)
```

At this point, gdb has our program loaded. Time to look at help:



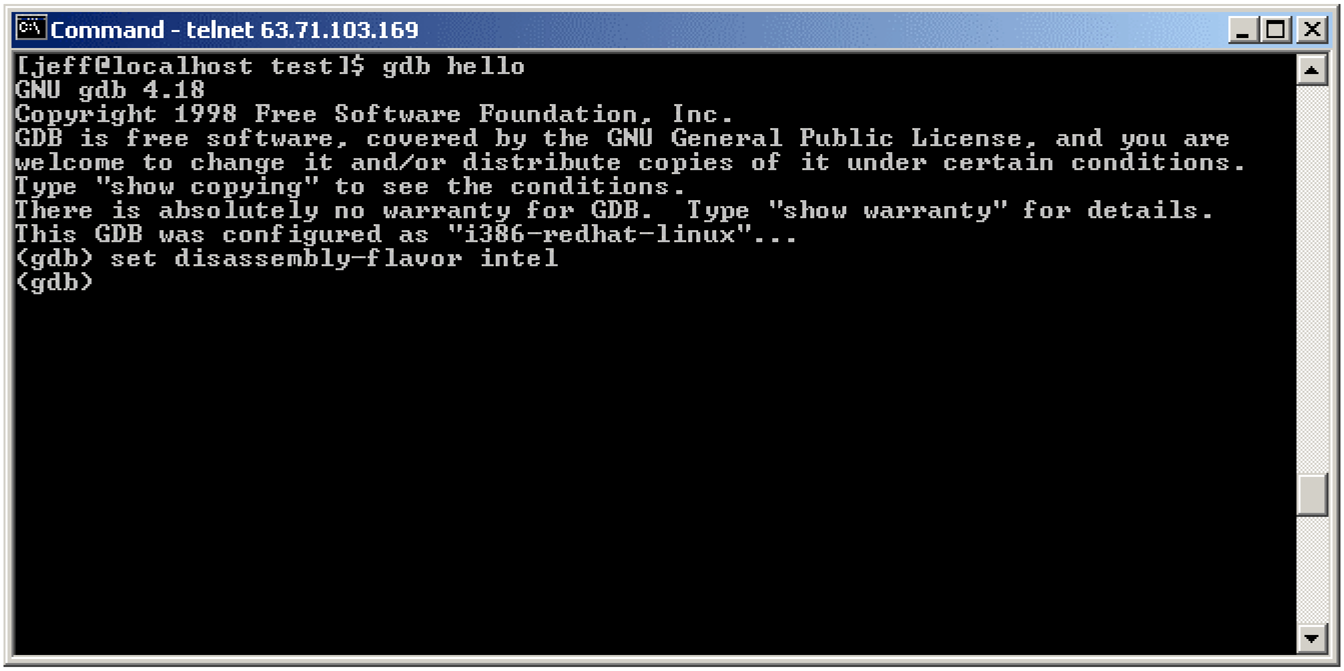
```
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(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)
```

The 'classes' of interest will be breakpoints, data, and stack.

To spare you the reading, here are some of the more useful commands:

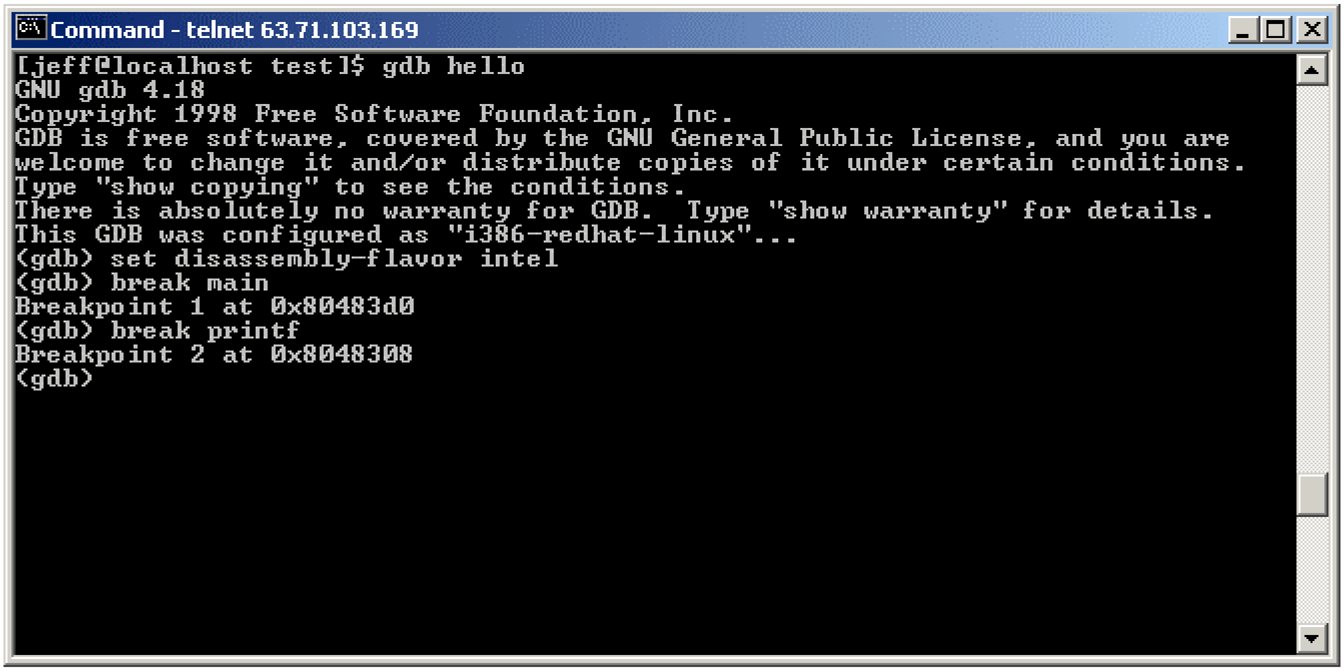
Function	Meaning
break 'function'	Sets a break point at entry to 'function'
delete	Deletes all break points
delete n	Deletes break point n
disassemble	Disassemble a specified section of memory. Default is the function surrounding the pc of the selected frame. With a single argument, the function surrounding that address is dumped. Two arguments are taken as a range of memory to dump.
print	Print value of expression EXP. Variables accessible are those of the lexical environment of the selected stack frame, plus all those whose scope is global or an entire file.
run	Start debugged program. You may specify arguments to give it. Args may include "*", or "[...]"; they are expanded using "sh". Input and output redirection with ">", "<", or ">>" are also allowed. With no arguments, uses arguments last specified (with "run" or "set args"). To cancel previous arguments and run with no arguments, use "set args" without arguments.
next	step into a function (see also 'help next' for a complete explanation)
step	step into a function (see also 'help step' for a complete explanation)
continue	continue execution
where	print the call stack (where you are in the program)
quit	exit gdb
info	info address -- Describe where symbol SYM is stored info all-registers -- List of all registers and their contents info args -- Argument variables of current stack frame info breakpoints -- Status of user-settable breakpoints info display -- Expressions to display when program stops info float -- Print the status of the floating point unit info frame -- All about selected stack frame info functions -- All function names info handle -- What debugger does when program gets various signals info line -- Core addresses of the code for a source line info locals -- Local variables of current stack frame info program -- Execution status of the program info registers -- List of integer registers and their contents info scope -- List the variables local to a scope info set -- Show all GDB settings info signals -- What debugger does when program gets various signals info source -- Information about the current source file info stack -- Backtrace of the stack info symbol -- Describe what symbol is at location ADDR info tracepoints -- Status of tracepoints info types -- All type names info variables -- All global and static variable names info watchpoints -- Synonym for "info breakpoints"

First thing is first. gdb's default assembly is AT&T (used by GAS, the GNU Assembler). Since we write with Intel assembly, we'll set that:



```
[jeff@localhost test]$ gdb hello
GNU gdb 4.18
Copyright 1998 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"...
(gdb) set disassembly-flavor intel
(gdb)
```

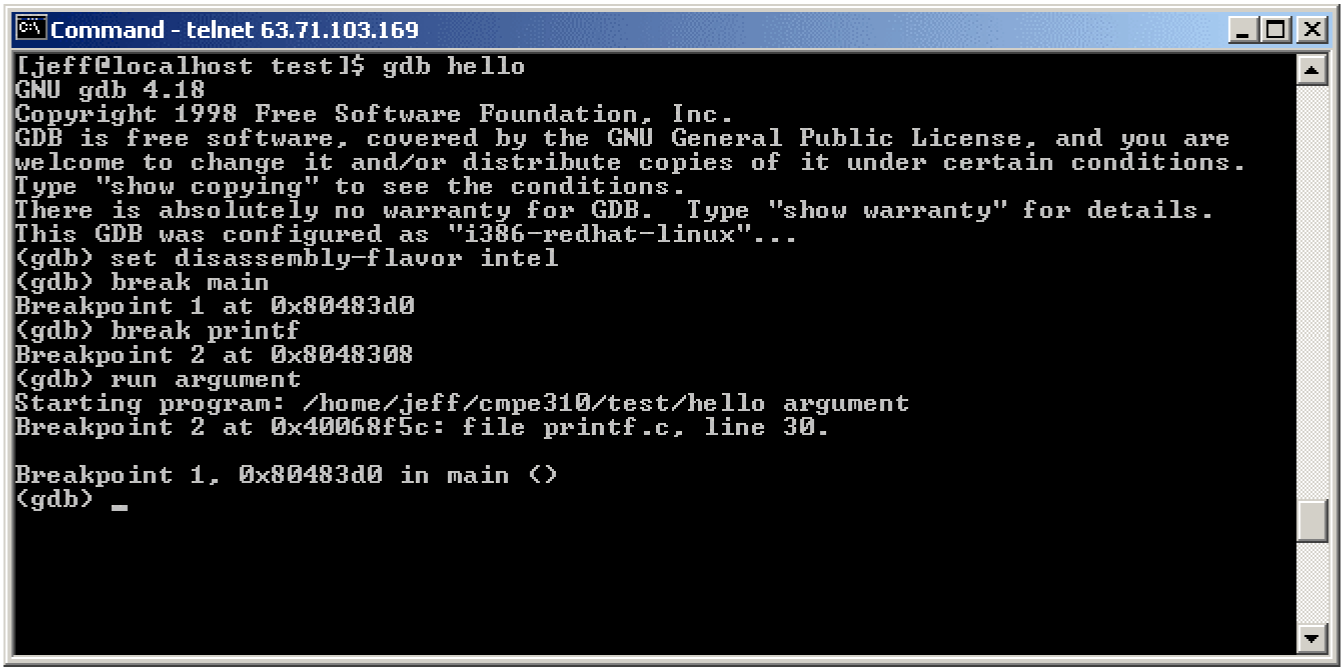
Set a break point in main and printf, shown below:



```
[jeff@localhost test]$ gdb hello
GNU gdb 4.18
Copyright 1998 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"...
(gdb) set disassembly-flavor intel
(gdb) break main
Breakpoint 1 at 0x80483d0
(gdb) break printf
Breakpoint 2 at 0x8048308
(gdb)
```

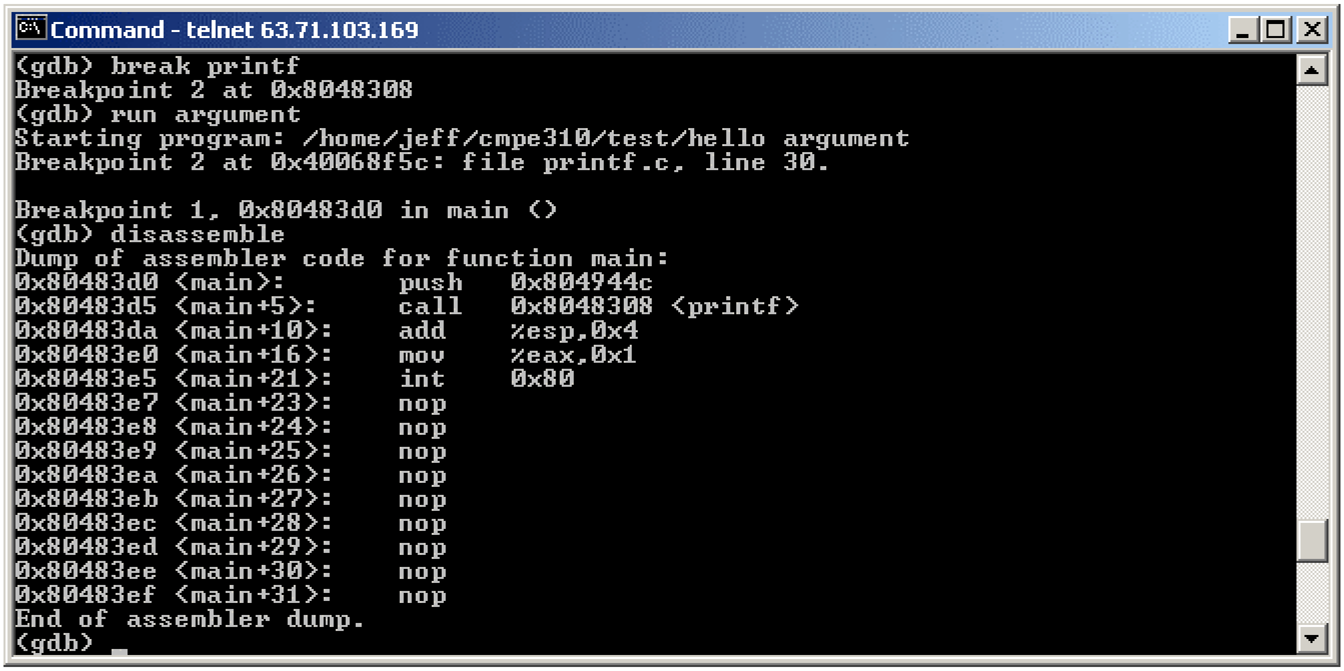
Breakpoint 1 is at memory address 0x80483d0, and 2 is at 0x8048308. To delete these break points, we could now issue 'delete' to remove all, 'delete 1' or 'delete 2' to remove a specific break point.

Finally, run the program. We'll run the program with an argument to see if we can find it later. Issue 'run argument':



```
[jeff@localhost test]$ gdb hello
GNU gdb 4.18
Copyright 1998 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"...
(gdb) set disassembly-flavor intel
(gdb) break main
Breakpoint 1 at 0x80483d0
(gdb) break printf
Breakpoint 2 at 0x8048308
(gdb) run argument
Starting program: /home/jeff/cmpe310/test/hello argument
Breakpoint 2 at 0x40068f5c: file printf.c, line 30.

Breakpoint 1, 0x80483d0 in main ()
(gdb) _
```

Not much here. Now would be a good time to issue 'disassemble':



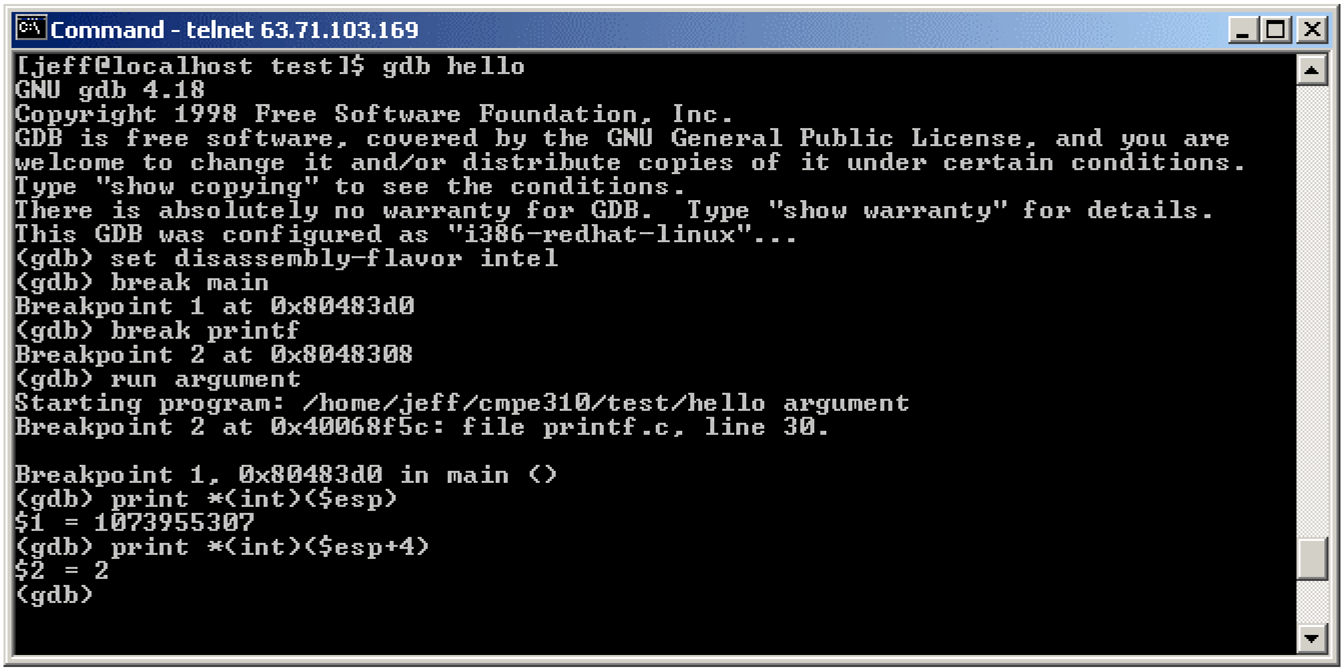
```
(gdb) break printf
Breakpoint 2 at 0x8048308
(gdb) run argument
Starting program: /home/jeff/cmpe310/test/hello argument
Breakpoint 2 at 0x40068f5c: file printf.c, line 30.

Breakpoint 1, 0x80483d0 in main ()
(gdb) disassemble
Dump of assembler code for function main:
0x80483d0 <main>:       push   0x804944c
0x80483d5 <main+5>:     call   0x8048308 <printf>
0x80483da <main+10>:    add    %esp,0x4
0x80483e0 <main+16>:    mov    %eax,0x1
0x80483e5 <main+21>:    int    0x80
0x80483e7 <main+23>:    nop
0x80483e8 <main+24>:    nop
0x80483e9 <main+25>:    nop
0x80483ea <main+26>:    nop
0x80483eb <main+27>:    nop
0x80483ec <main+28>:    nop
0x80483ed <main+29>:    nop
0x80483ee <main+30>:    nop
0x80483ef <main+31>:    nop
End of assembler dump.
(gdb) _
```

Seems we've lost much of our debug information. This is due to nasm. nasm has not left us much, but its enough we can work with.

We know we passed a command line argument to the program. We'll try to find it. The stack should look similar to below:

esp + C	char* env[]	pointer
esp + 8	char* argv[]	pointer
esp + 4	argc	integer
esp →	???	unknown

gdb and dbx have very powerful expression evaluators. We'll dig for argc. It should be 2:



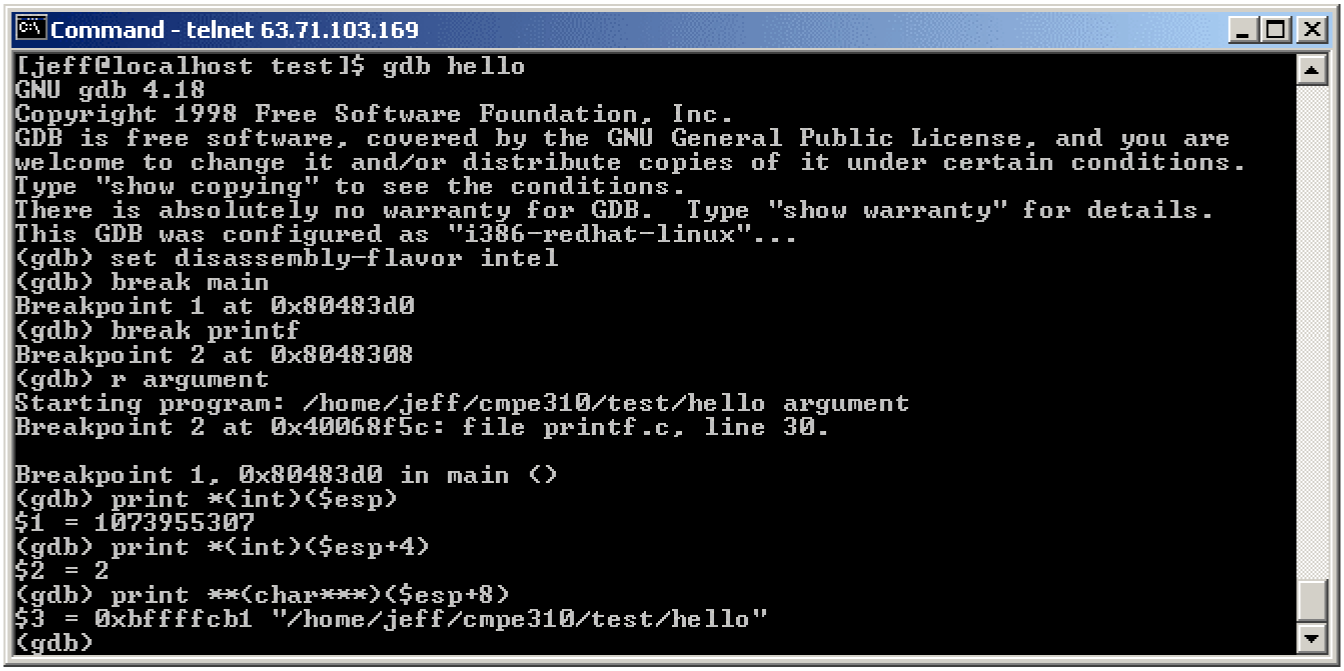
```
[jeff@localhost test]$ gdb hello
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) set disassembly-flavor intel
(gdb) break main
Breakpoint 1 at 0x80483d0
(gdb) break printf
Breakpoint 2 at 0x8048308
(gdb) run argument
Starting program: /home/jeff/cmpe310/test/hello argument
Breakpoint 2 at 0x40068f5c: file printf.c, line 30.

Breakpoint 1, 0x80483d0 in main ()
(gdb) print *(int)($esp)
$1 = 1073955307
(gdb) print *(int)($esp+4)
$2 = 2
(gdb)
```

It seems we found argc at esp + 4. Here's what we did:

- When printing a register, prefix the register name with a '$'
- esp + 4 is an address. This required a dereference '*'
- The argument was an integer. Cast it as such 'int'

So, the final expression was *(int)($esp+4)

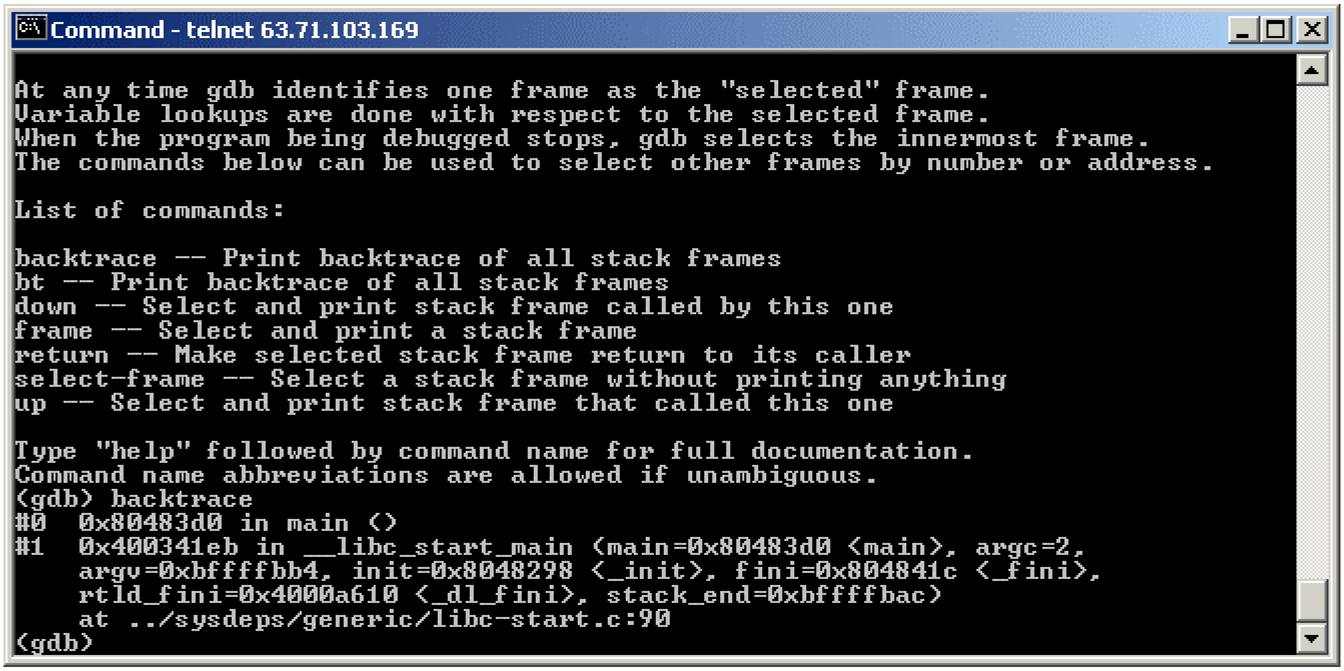
Easy enough. Lets poke around and find the program name:



```
[jeff@localhost test]$ gdb hello
GNU gdb 4.18
Copyright 1998 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"...
(gdb) set disassembly-flavor intel
(gdb) break main
Breakpoint 1 at 0x80483d0
(gdb) break printf
Breakpoint 2 at 0x8048308
(gdb) r argument
Starting program: /home/jeff/cmpe310/test/hello argument
Breakpoint 2 at 0x40068f5c: file printf.c, line 30.

Breakpoint 1, 0x80483d0 in main ()
(gdb) print *(int)($esp)
$1 = 1073955307
(gdb) print *(int)($esp+4)
$2 = 2
(gdb) print **(char***)($esp+8)
$3 = 0xbffffcb1 "/home/jeff/cmpe310/test/hello"
(gdb)
```

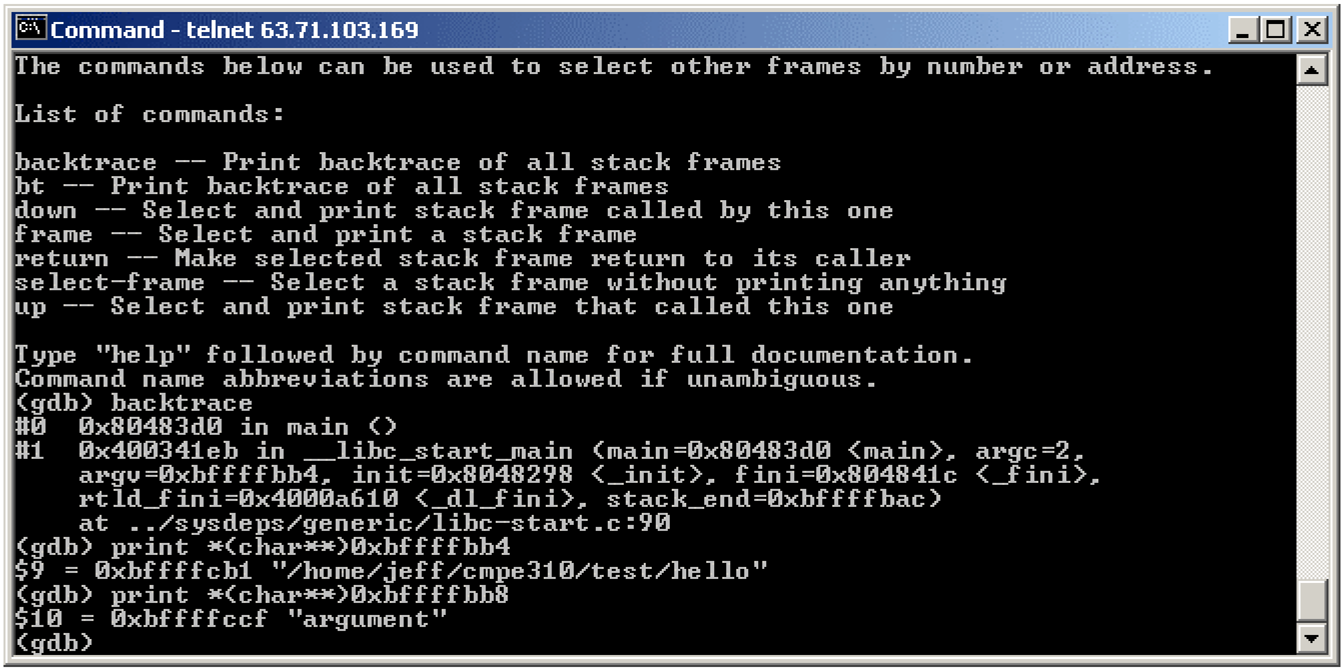
This was a little tougher. Basically, argv[] is a char**. At esp + 8, we found a pointer to the char**. So, we needed to double dereference to get the char* (argv[0]). argv[1] will be found similarly with an expression such as 'print *(* (char*) ((char**)($esp+8) + 4)'. Basically, you will add 4 (bump the pointer) before the final dereference

Another way to find argv[] is to issue 'backtrace' while in main:



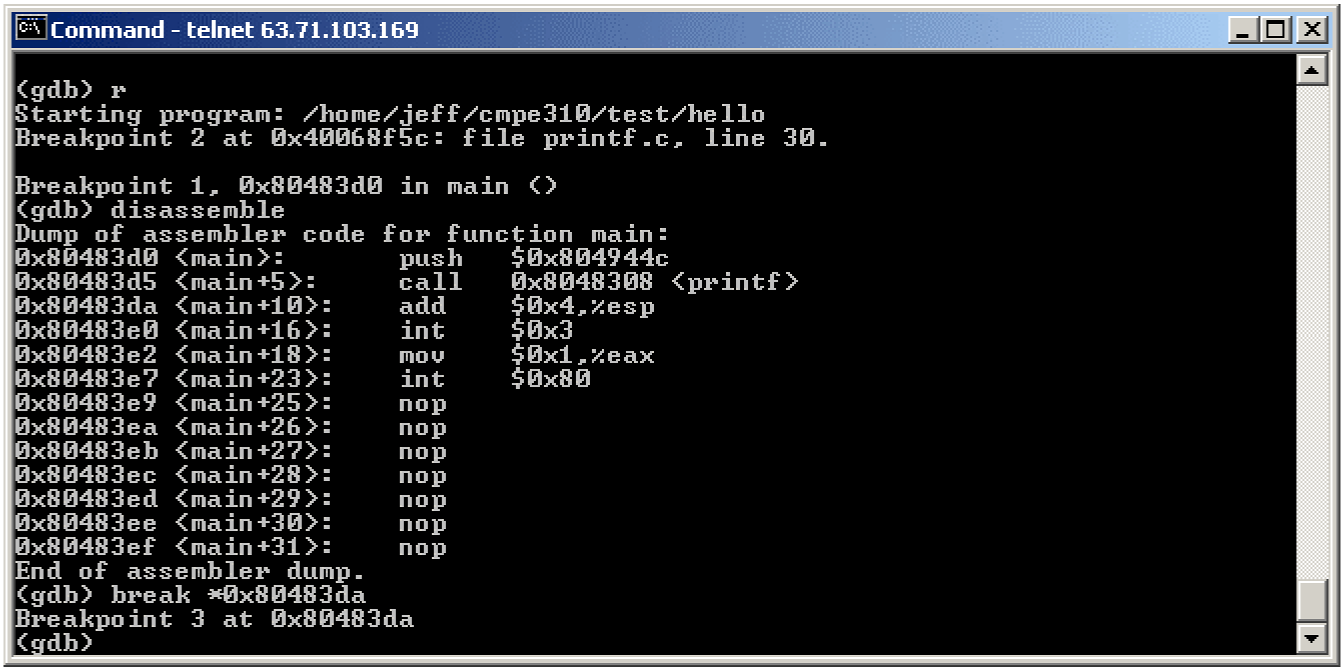
```

At any time gdb identifies one frame as the "selected" frame.
Variable lookups are done with respect to the selected frame.
When the program being debugged stops, gdb selects the innermost frame.
The commands below can be used to select other frames by number or address.

List of commands:

backtrace -- Print backtrace of all stack frames
bt -- Print backtrace of all stack frames
down -- Select and print stack frame called by this one
frame -- Select and print a stack frame
return -- Make selected stack frame return to its caller
select-frame -- Select a stack frame without printing anything
up -- Select and print stack frame that called this one

Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb) backtrace
#0  0x80483d0 in main ()
#1  0x400341eb in __libc_start_main (main=0x80483d0 <main>, argc=2,
    argv=0xbffffbb4, init=0x8048298 <_init>, fini=0x804841c <_fini>,
    rtld_fini=0x4000a610 <_dl_fini>, stack_end=0xbffffbac)
    at ../sysdeps/generic/libc-start.c:90
(gdb)
```

argv[0] is at 0xbffffbb4. argv[1] will be at 0xbffffbb8:



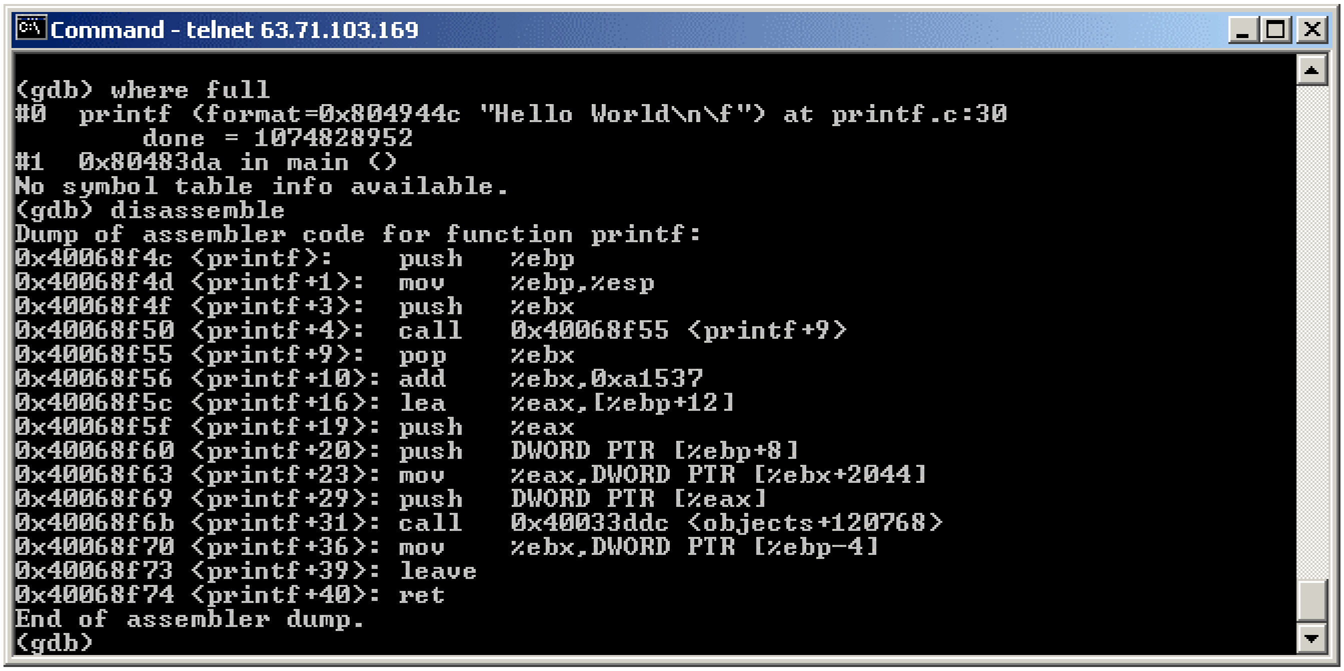
```
The commands below can be used to select other frames by number or address.

List of commands:

backtrace -- Print backtrace of all stack frames
bt -- Print backtrace of all stack frames
down -- Select and print stack frame called by this one
frame -- Select and print a stack frame
return -- Make selected stack frame return to its caller
select-frame -- Select a stack frame without printing anything
up -- Select and print stack frame that called this one

Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb) backtrace
#0  0x80483d0 in main ()
#1  0x400341eb in __libc_start_main (main=0x80483d0 <main>, argc=2,
    argv=0xbffffbb4, init=0x8048298 <_init>, fini=0x804841c <_fini>,
    rtld_fini=0x4000a610 <_dl_fini>, stack_end=0xbffffbac)
    at ../sysdeps/generic/libc-start.c:90
(gdb) print *(char**)0xbffffbb4
$9 = 0xbffffcb1 "/home/jeff/cmpe310/test/hello"
(gdb) print *(char**)0xbffffbb8
$10 = 0xbffffccf "argument"
(gdb)
```

Enough fooling around with argc and argv[]. Set a breakpoint to stop after the call to printf . Issue 'break *0x80483da' (substitute the address as required).



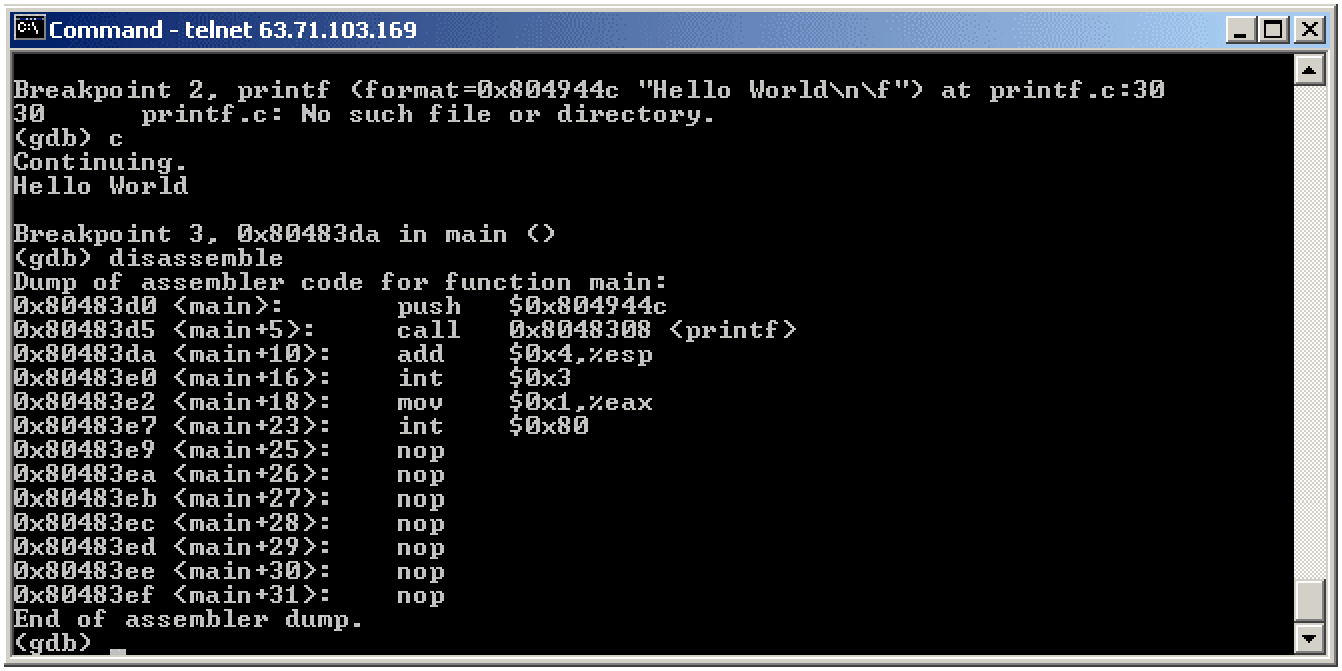
```

(gdb) r
Starting program: /home/jeff/cmpe310/test/hello
Breakpoint 2 at 0x40068f5c: file printf.c, line 30.

Breakpoint 1, 0x80483d0 in main ()
(gdb) disassemble
Dump of assembler code for function main:
0x80483d0 <main>:       push   $0x804944c
0x80483d5 <main+5>:     call   0x8048308 <printf>
0x80483da <main+10>:    add    $0x4,%esp
0x80483e0 <main+16>:    int    $0x3
0x80483e2 <main+18>:    mov    $0x1,%eax
0x80483e7 <main+23>:    int    $0x80
0x80483e9 <main+25>:    nop
0x80483ea <main+26>:    nop
0x80483eb <main+27>:    nop
0x80483ec <main+28>:    nop
0x80483ed <main+29>:    nop
0x80483ee <main+30>:    nop
0x80483ef <main+31>:    nop
End of assembler dump.
(gdb) break *0x80483da
Breakpoint 3 at 0x80483da
(gdb)
```

Issue 'continue' to start execution. We hit the second break point in printf. Another useful commands at this point is 'where' to get our call stack. We also get our format string since we are in printf.



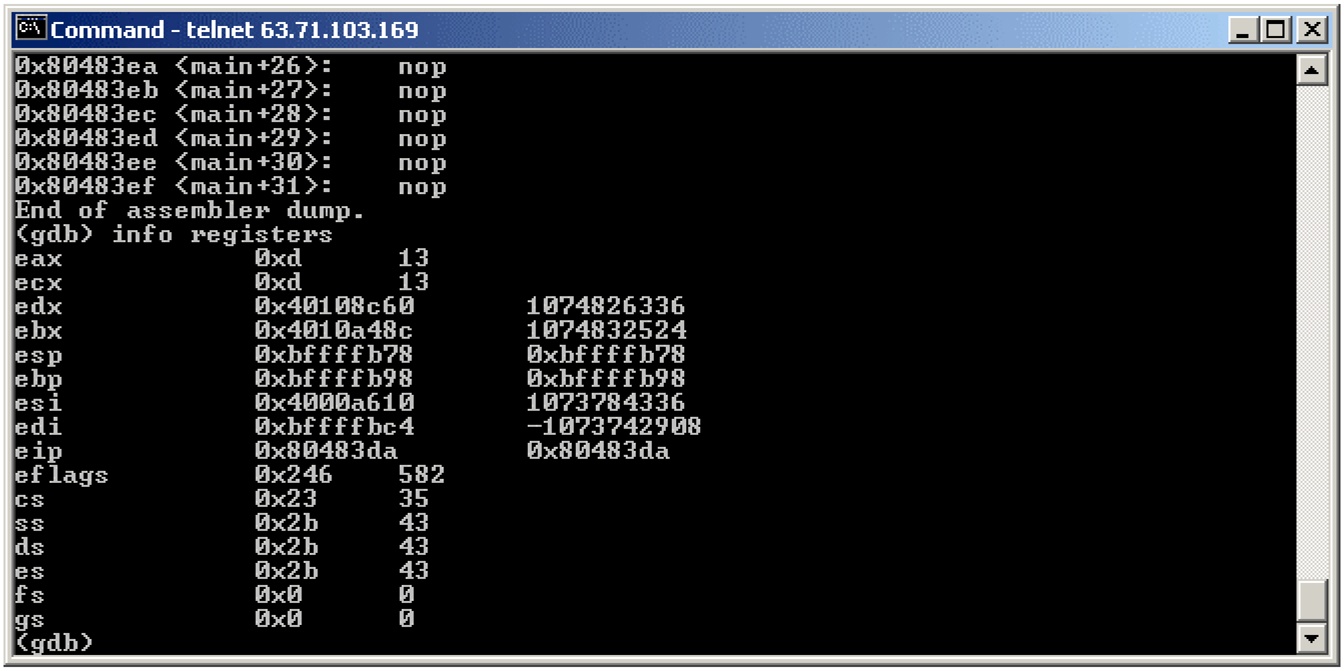
```
(gdb) where full
#0  printf (format=0x804944c "Hello World\n\f") at printf.c:30
        done = 1074828952
#1  0x80483da in main ()
No symbol table info available.
(gdb) disassemble
Dump of assembler code for function printf:
0x40068f4c <printf>:    push   %ebp
0x40068f4d <printf+1>:  mov    %ebp,%esp
0x40068f4f <printf+3>:  push   %ebx
0x40068f50 <printf+4>:  call   0x40068f55 <printf+9>
0x40068f55 <printf+9>:  pop    %ebx
0x40068f56 <printf+10>: add    %ebx,0xa1537
0x40068f5c <printf+16>: lea    %eax,[%ebp+12]
0x40068f5f <printf+19>: push   %eax
0x40068f60 <printf+20>: push   DWORD PTR [%ebp+8]
0x40068f63 <printf+23>: mov    %eax,DWORD PTR [%ebx+2044]
0x40068f69 <printf+29>: push   DWORD PTR [%eax]
0x40068f6b <printf+31>: call   0x40033ddc <objects+120768>
0x40068f70 <printf+36>: mov    %ebx,DWORD PTR [%ebp-4]
0x40068f73 <printf+39>: leave
0x40068f74 <printf+40>: ret
End of assembler dump.
(gdb)
```

Continue once again, and we break at address 0x80483da. We just returned from printf.



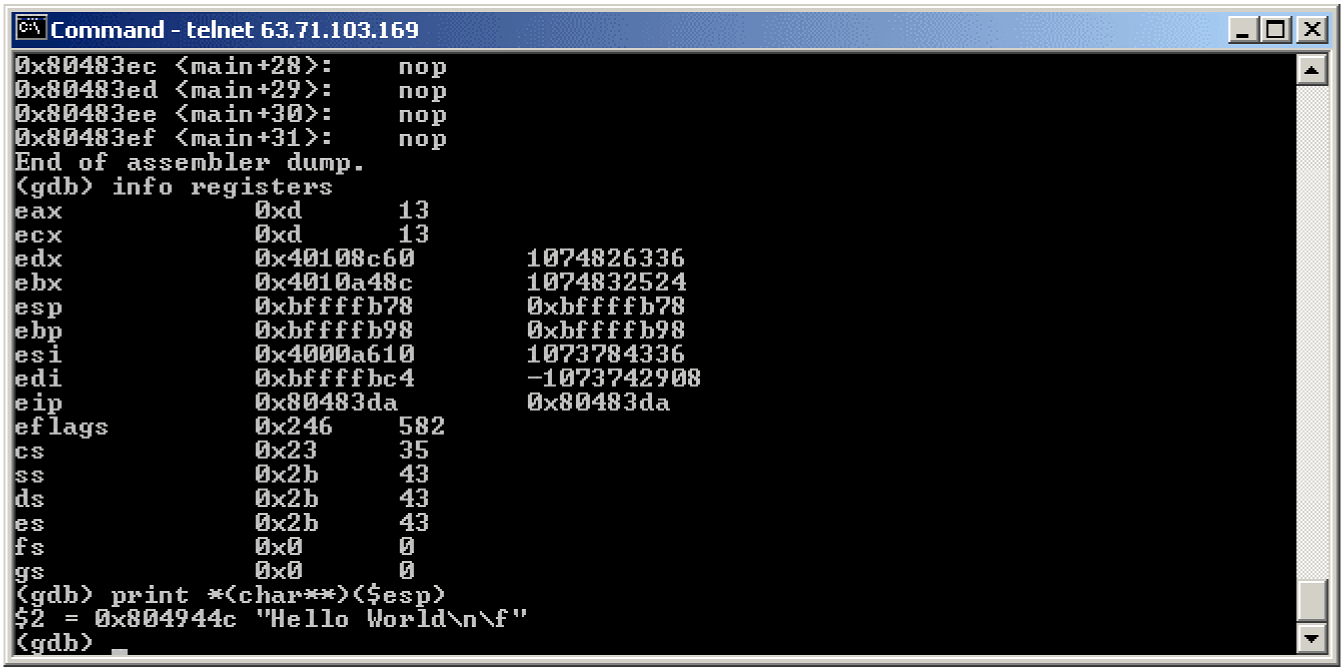
```
Breakpoint 2, printf (format=0x804944c "Hello World\n\f") at printf.c:30
30      printf.c: No such file or directory.
(gdb) c
Continuing.
Hello World

Breakpoint 3, 0x80483da in main ()
(gdb) disassemble
Dump of assembler code for function main:
0x80483d0 <main>:       push   $0x804944c
0x80483d5 <main+5>:     call   0x8048308 <printf>
0x80483da <main+10>:    add    $0x4,%esp
0x80483e0 <main+16>:    int    $0x3
0x80483e2 <main+18>:    mov    $0x1,%eax
0x80483e7 <main+23>:    int    $0x80
0x80483e9 <main+25>:    nop
0x80483ea <main+26>:    nop
0x80483eb <main+27>:    nop
0x80483ec <main+28>:    nop
0x80483ed <main+29>:    nop
0x80483ee <main+30>:    nop
0x80483ef <main+31>:    nop
End of assembler dump.
(gdb) _
```

Issue 'info registers' to see what's in the registers:



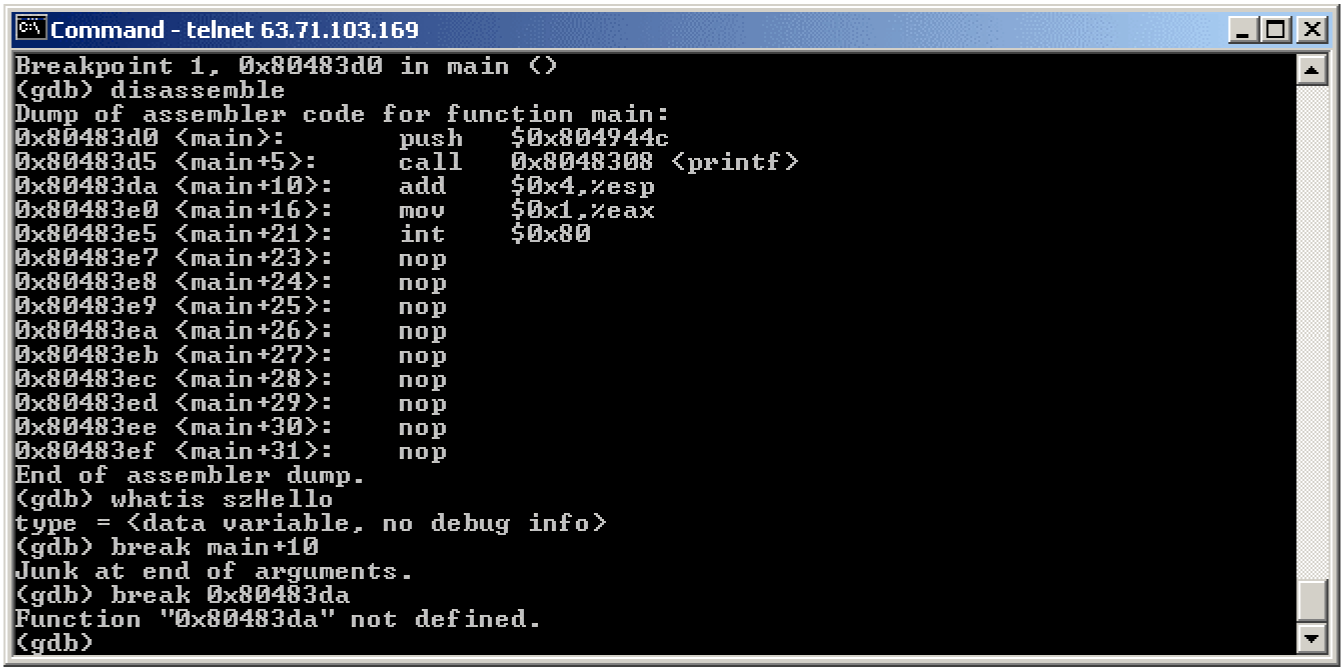
```
0x80483ea <main+26>:    nop
0x80483eb <main+27>:    nop
0x80483ec <main+28>:    nop
0x80483ed <main+29>:    nop
0x80483ee <main+30>:    nop
0x80483ef <main+31>:    nop
End of assembler dump.
(gdb) info registers
eax            0xd      13
ecx            0xd      13
edx            0x40108c60       1074826336
ebx            0x4010a48c       1074832524
esp            0xbffffb78       0xbffffb78
ebp            0xbffffb98       0xbffffb98
esi            0x4000a610       1073784336
edi            0xbffffbc4       -1073742908
eip            0x80483da        0x80483da
eflags         0x246    582
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x0      0
(gdb)
```

Its interesting to see what is in eax and ecx. Could this be the CR (carriage return) that ended our string? We have not cleaned the stack yet. esp should point to the string we just printed. Issue 'print *(char**)($esp)':



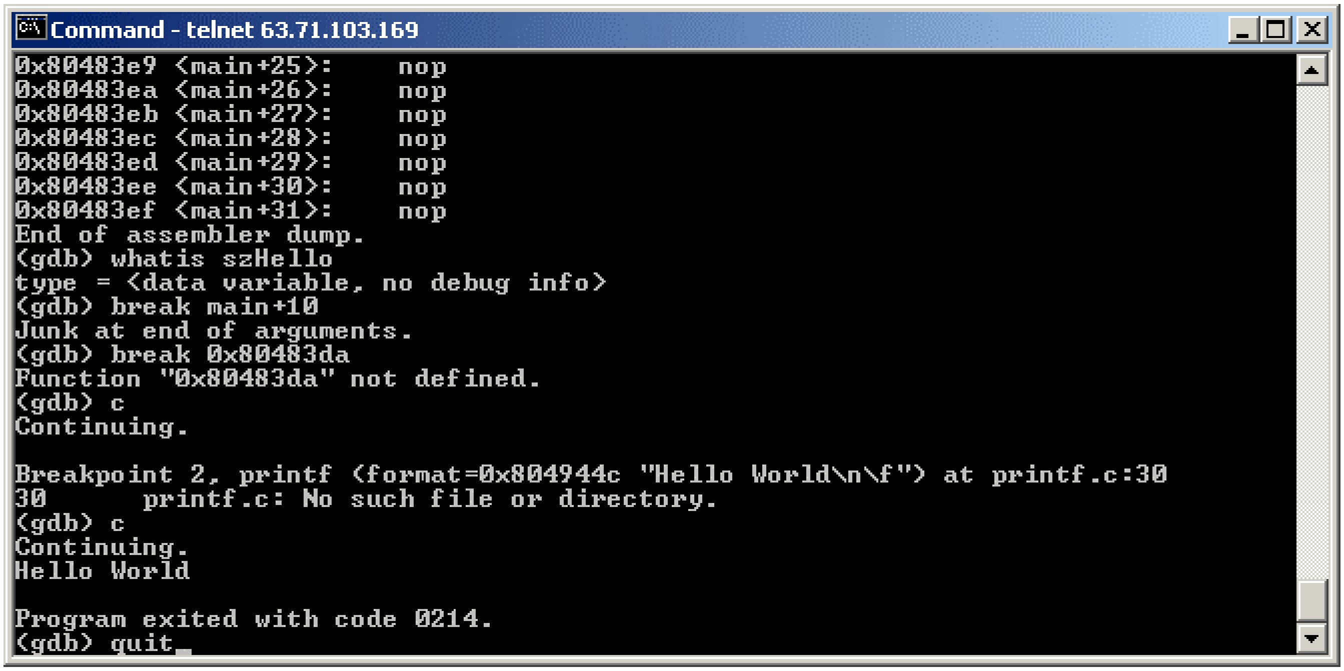
```
0x80483ec <main+28>:    nop
0x80483ed <main+29>:    nop
0x80483ee <main+30>:    nop
0x80483ef <main+31>:    nop
End of assembler dump.
(gdb) info registers
eax            0xd      13
ecx            0xd      13
edx            0x40108c60       1074826336
ebx            0x4010a48c       1074832524
esp            0xbffffb78       0xbffffb78
ebp            0xbffffb98       0xbffffb98
esi            0x4000a610       1073784336
edi            0xbffffbc4       -1073742908
eip            0x80483da        0x80483da
eflags         0x246    582
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x0      0
(gdb) print *(char**)($esp)
$2 = 0x804944c "Hello World\n\f"
(gdb) _
```

Other useful commands (that don't work due to nasm's lack of debug information) are 'next', 'step', 'xbreak', and 'whatis' which gives you type information. Also, note that you must use an '*' to specify a break on an address:



```
Breakpoint 1, 0x80483d0 in main ()
(gdb) disassemble
Dump of assembler code for function main:
0x80483d0 <main>:       push   $0x804944c
0x80483d5 <main+5>:     call   0x8048308 <printf>
0x80483da <main+10>:    add    $0x4,%esp
0x80483e0 <main+16>:    mov    $0x1,%eax
0x80483e5 <main+21>:    int    $0x80
0x80483e7 <main+23>:    nop
0x80483e8 <main+24>:    nop
0x80483e9 <main+25>:    nop
0x80483ea <main+26>:    nop
0x80483eb <main+27>:    nop
0x80483ec <main+28>:    nop
0x80483ed <main+29>:    nop
0x80483ee <main+30>:    nop
0x80483ef <main+31>:    nop
End of assembler dump.
(gdb) whatis szHello
type = <data variable, no debug info>
(gdb) break main+10
Junk at end of arguments.
(gdb) break 0x80483da
Function "0x80483da" not defined.
(gdb)
```

And finally, issue 'quit' to exit the program:



```
0x80483e9 <main+25>:    nop
0x80483ea <main+26>:    nop
0x80483eb <main+27>:    nop
0x80483ec <main+28>:    nop
0x80483ed <main+29>:    nop
0x80483ee <main+30>:    nop
0x80483ef <main+31>:    nop
End of assembler dump.
(gdb) whatis szHello
type = <data variable, no debug info>
(gdb) break main+10
Junk at end of arguments.
(gdb) break 0x80483da
Function "0x80483da" not defined.
(gdb) c
Continuing.

Breakpoint 2, printf (format=0x804944c "Hello World\n\f") at printf.c:30
30      printf.c: No such file or directory.
(gdb) c
Continuing.
Hello World

Program exited with code 0214.
(gdb) quit_
```

Also note that you can code an 'int 3' directly in your source if you want to stop while under the debugger. This way, you don't have to place a breakpoint on the command line.