The following is provided courtesy of Dr. Richard Chang

Using gdb for Assembly Language Debugging

Introduction

You may have used the GNU debugger gdb to debug C/C++ programs in CMSC 201 and 202. For this course, you will also use gdb to debug assembly language programs. Using a debugger is more important for assembly language programming than a high-level programming language because printing out values from an assembly language program is non-trivial, hence adding debugging code to an assembly language program might itself introduce new bugs, or at least make the bug behave differently.

If your program contains portions written in C, those portions should be compiled using the -g option to retain debugging information in the a.out file. Although the NASM documentation describes a similar -g option for NASM, this option doesn't do anything when the output is in ELF format.

The complete documentation for gdb is available on the Linux system using the "info" command:

```
   info gdb
```

We will summarize some of the more useful commands for assembly language debugging below.

Starting and stopping programs

After you've assembled and linked your program using nasm and ld, you can invoke the debugger using the UNIX command:

```
   gdb a.out
```

At this point gdb might complain that "no debugging symbols found". Just ignore this particular comment. You can now run your program inside gdb by typing "run" and the gdb prompt. However, the program will just behave exactly as it did when it ran under the Linux shell. The point of using a debugger is to set "breakpoints" where the execution of the program is halted. This returns you to the gdb prompt and you can use various gdb commands to examine the contents of the registers and/or memory. It is common to have breakpoints set at the beginning of a program, at the end of a program, at the top of a loop and anywhere you think the bug occurred in your program. When using gdb with C/C++ programs, you can set breakpoints by the line number of the statement in the source file. However, NASM doesn't generate the line number information for gdb, so you must set the breakpoints according to addresses relative to labels in the assembly language program.

For example:

```
   break *_start+5
```

Stops the execution of the program 5 bytes after the address labeled by _start. Due to a bug in gdb,

```
   break *_start
```

does not stop the program before the execution of the first instruction. A workaround is to add a no operation instruction (mnemonic "nop") to the beginning of your program and use:

```
   break *_start+1
```

After execution of your program has been halted by a breakpoint, you can resume execution in several ways. The "cont" (continue) command resumes execution after the breakpoint. The execution continues until the next breakpoint is reached or the program terminates. Alternatively the single step commands "stepi" or "nexti" command may be used to execute a single machine instruction after the breakpoint. The difference between "stepi" and "nexti" arises when the next instruction is a function call. The "nexti" command will continue execution until the return from the function call (which might be never). On the other hand, the "stepi" command will enter the function. The command "where" will show where the execution of a program has halted.

To help you determine where to set the breakpoints, gdb also has a disassembler. The command:

```
disassemble _start
```

will print out the mnemonics for the machine instructions starting at the label _start. Since the GNU assembler uses AT&T style syntax, gdb also uses AT&T style syntax instead of Intel-style syntax. Most importantly, in AT&T style syntax the instruction for loading register EAX with the constant 1 looks like

```
     mov     0x1, %eax
```

In Intel-style syntax, the order of the source and destination are reversed. Recent versions of gdb allow you to switch the order of the source and destination during disassembly with the command:

```
set disassembly-flavor intel
```

You can configure gdb to always use Intel-style syntax by including the command above in a file named .gdbinit in your home directory.

Examining register and memory contents

Using a combination of breakpoints and single-stepping, you should be able to trace through the sections of your code that you suspect to be buggy. This allows you to look at the contents of the registers and memory when the execution of the program has halted. The command:

```
info registers
```

prints out the contents of every register, including all the segment registers. This is often too much information. The "print" command is much more useful. For example, the commands:

```
print/d $ecx
print/x $ecx
print/t $ecx
```

print out the contents of the ECX register in decimal, hexadecimal, and binary, respectively.

Memory contents can be examined using the "x" command. The "x" command is optionally followed by a "/", a count field, a format field, a size field and finally a memory address. The count field is a number in decimal. The format field is a single letter with 'd' for decimal, 'x' for hexadecimal, 't' for binary and 'c' for ASCII. The size field is also a single letter with 'b' for byte, 'h' for 16-bit word (half word) and 'w' for a 32-bit word. For example, if your program has a label "msg", the following commands:

```
x/12cb &msg
x/12db &msg
x/12xh &msg
x/12xw &msg
```

will print out respectively the contents of memory starting at msg in the following manner: the next 12 bytes as ASCII characters, the next 12 bytes as decimal numbers, the next 12 16-bit words in hex, and the next 12 32-bit words in hex.

Sometimes, as you are tracing your program, you are really interested in the contents of a specific register. If you issue the command:

```
   display $eax
```

then the contents of the EAX register will be printed to the screen every time the program is halted. This saves a lot of typing. You may issue more than one display command and you can use format codes like "/x" to output the values in hex. The gdb command "info display" will list all the active displays. Use "undisplay" to remove an item on this list.

A very good use of the display command is to have the next instruction of the program printed whenever the program is halted:

```
  display/i $eip
```

Command Summary

We've only covered a handful of useful gdb commands in this tutorial to get you started using gdb. Many of these commands can be shortened (e.g., "si" is equivalent to "stepi"). The gdb debugger itself has a nice help feature. Simply type "help" to get a list of commands, and

```
  help [command]
```

will give you more information on that command. Again, more complete documentation on gdb is available through the "info gdb" command typed in the UNIX shell.

Command	**Example**	**Description**
run		start program
quit		quit out of gdb
cont		continue execution after a break
break [addr]	break *_start+5	sets a breakpoint
delete [n]	delete 4	removes nth breakpoint
delete		removes all breakpoints
info break		lists all breakpoints
stepi		execute next instruction
stepi [n]	stepi 4	execute next n instructions
nexti		execute next instruction, stepping over function calls
nexti [n]	nexti 4	execute next n instructions, stepping over function calls
where		show where execution halted

disas [addr]	disas _start	disassemble instructions at given address
info registers		dump contents of all registers
print/d [expr]	print/d $ecx	print expression in decimal
print/x [expr]	print/x $ecx	print expression in hex
print/t [expr]	print/t $ecx	print expression in binary
x/NFU [addr]	x/12xw &msg	Examine contents of memory in given format
display [expr]	display $eax	automatically print the expression each time the program is halted
info display		show list of automatically displays
undisplay [n]	undisplay 1	remove an automatic display