

=====
LINUX ASSEMBLER TUTORIAL

by

Robin Miyagi

@

<http://www.geocities.com/SiliconValley/Ridge/2544/>
=====

start@: Thu Feb 03 02:14:37 UTC 2000

update: Fri Jul 30 23:52:23 UTC 2000

update: Fri Sep 15 22:39:17 UTC 2000 :

- This tutorial now explains Linux assembler in terms of the assembler `as'.

- Information about Binutils programs such as Objdump, and Discussion on Debugging and `gdb' is ad

update: Thu Jan 11 20:13:06 UTC 2001 :

=====
* Introduction

When programming in assembler for Linux (or any other Unix var for that matter), it is important to remember that Linux is a protected mode operating system (on i386 machines, Linux operates the CPU in protected mode). This means that ordinary user processes are not allowed to do certain things, such as access or access IO ports. Writing Linux kernel modules on the other (which operate in kernel mode), are allowed to access hardware directly (Read the Assembler-HOWTO on my assembler page for information on this issue). User mode processes may access hardware using device files. Device files actually access kernel modules which access hardware directly. This file will be restricted to user mode operation. See my pages on kernel module programming.

Please email me comments and suggestions regarding this tutorial at penguin@dccnet.com.

* System Calls

In programming in assembler for DOS you probably made use

software interrupts, especially the int 0x21 functions which are the DOS system calls. In Linux, system calls are made via int 0x80. The system call number is passed via register EAX, and the parameters to the system call are passed via the remaining registers. This discussion only applies if there are no more than five parameters passed to the system call. If there are more than 5 parameters, the parameters must be located in memory (e.g. on the stack), and EBX must contain the address of the beginning of the parameters.

If you would like a list of the system call numbers, look at the contents of /usr/include/asm/unistd.h. If you would like information about a specific system call (e.g. write ()), type 'man 2 write' at the prompt. Section 2 of the linux man pages covers system calls.

If you look at the contents of /usr/include/asm/unistd.h, you will see the following line near the top of the file;

```
#define __NR_write 4
```

This indicates that register EAX must be set to 4 in order to execute the write () system call. Now, if you execute the following command;

```
$ man 2 write
```

you get the following function description (under the SYNOPSIS heading).

```
ssize_t write(int fd, const void *buf, size_t count);
```

This indicates that ebx is equal to the file descriptor of the file you want to write to, ecx is a pointer to the string you want to write, and edx contains the length of the string. If there are more parameters to this system call, they would be placed in edi and esi respectively.

How do I know the file descriptor for stdout is 1. If you look in your /dev directory, you will notice that /dev/stdout is a symbolic link that points to /proc/self/fd/1. Therefore stdout is file descriptor 1.

I leave looking up the _exit system call as an exercise.

In linux, system calls are processed by the kernel.

* GNU Assembler

On most Linux systems, you will usually find the GNU C compiler (gcc). This compiler uses an assembler called 'as' as a back-

This means that the C compiler translates the C code into assembly which in turn is assembled by `as' to an object file (*.o).

`As' uses the AT&T syntax. Experienced intel syntax assembly programmers find AT&T `really weird'. It is really no more or less difficult than intel syntax. I switched over to `as' because there is less ambiguity, works better with the standard GNU/Linux programs such as gdb (supports the gstacks format), objdump (objdump disassembles code in `as' syntax). In short, it is a standard component of a GNU Linux system with programming tools installed will explain debugging and objdump later in this tutorial.

If you would like more information about `as' look in the documentation under as (e.g. type `info as' at the shell prompt). Also look in the info documentation on the Binutils package (package contains such programming tools as objdump, ld, etc.).

** GNU assembler v.s. Intel Syntax

Since most assembler documentation for the i386 platform is written using intel syntax, some comparison between the 2 formats is in order. Here is a summarized list of the differences;

- In `as' the source comes before the destination, opposite the intel syntax.
- The opcodes are suffixed with a letter indicating the size of the operands (e.g. `l' for dword, `w' for word, `b' for byte).
- Immediate values must be prefixed with a `\$', and registers must be prefixed with a `%'.
Effective addresses use the General Syntax DISP(BASE,INDEX,SCALE). A concrete example would be;

```
movl mem_location(%ebx,%ecx,4), %eax
```

Which is equivalent to the following in intel syntax;

```
mov eax, [eax + ecx*4 + mem_location]
```

Now for an example illustrating the difference (intel version comments);

```
movl %eax, %ebx          # mov %ebx, %eax
movw $0x3c4a, %ax
```

Now for our little program;

```

## hello-world.s

## by Robin Miyagi
## http://www.geocities.com/SiliconValley/Ridge/2544/

## Compile Instructions:
## -----
## as -o hello-world.o hello-world.s
## ld -o hello-world -O0 hello-world.o

## This file is a basic demonstration of the GNU assembl
## `as'.

## This program displays a friendly string on the screen u
## the write () system call
#####
.section .data
hello:
    .ascii "Hello, world!\n"
hello_len:
    .long    . - hello
#####
.section .text
    .globl _start

_start:
## display string using write () system call
xorl %ebx, %ebx        # %ebx = 0
movl $4, %eax          # write () system call
xorl %ebx, %ebx        # %ebx = 0
incl %ebx              # %ebx = 1, fd = stdout
leal hello, %ecx       # %ecx ---> hello
movl hello_len, %edx   # %edx = count
int $0x80              # execute write () system call

## terminate program via _exit () system call
xorl %eax, %eax        # %eax = 0
incl %eax              # %eax = 1 system call _exit ()
xorl %ebx, %ebx        # %ebx = 0 normal program return cod
int $0x80              # execute system call _exit ()

```

In the above program, notice the use of `#' to start comments. also supports the `/* C comment */' syntax. If you use the C comment syntax, it works exactly the same as for C (multiple lines, as as inline commenting). I always use the `#' comment syntax, as works better with emacs' asm-mode. The double `##' is allowed not necessary (this is only because of a quirk of emacs asm-mod

Notice the names of the sections .text, and .data. these are

in ELF files to tell the linker where the code and data segments are. There is also the .bss section to store uninitialized data. It is only these sections that occupy memory during program execution.

* Accessing Command Line Arguments and Environment Variables

When an ELF executable starts running, the command line arguments and environment variables are available on the stack. In assembly this means that you may access these via the pointer stored in memory when the program starts execution. See the documentation on assembler programming page relating to the ELF binary format.

So how is this data arranged on the stack? Quite simple really. The number of command line arguments (including the name of program) are stored as an integer at [esp]. Then, at [esp+4] a pointer to the first command line argument (which is the name of program) is stored. If there were any additional command line parameters, their pointers would be stored in [esp+8], [esp+12], etc. After all the command line argument pointers, comes a NULL pointer. After the NULL pointer are all the pointers to environment variables, and then finally a NULL pointer to indicate the end of the environment variables have been reached.

A summary of the initial ELF stack is shown below;

```
(%esp)      argc, count of arguments (integer)
4(%esp)     char *argv (pointer to first command line argument)
...         pointers to the rest of the command line arguments
?(%esp)     NULL pointer
...         pointers to environment variables
??(%esp)    NULL pointer
```

Now for our little program;

```
-----
## stack-param.s #####

## Robin Miyagi #####
## http://www.geocities.com/SiliconValley/Ridge/2544/ #####

## This file shows how one can access command line parameters
## via the stack at process start up. This behavior is defined
## in the ELF specification.

## Compile Instructions:
## -----
## as -o stack-param.o stack-param.s
## ld -O0 -o stack-param stack-param.o
#####
.section .data
```

```

new_line_char:
    .byte 0x0a
#####
    .section .text

    .globl _start

    .align 4
_start:
    movl %esp, %ebp        # store %esp in %ebp
again:
    addl $4, %esp          # %esp ---> next parameter on stack
    movl (%esp), %eax      # move next parameter into %eax
    testl %eax, %eax       # %eax (parameter) == NULL pointer?
    jz end_again           # get out of loop if yes
    call putstring         # output parameter to stdout.
    jmp again              # repeat loop
end_again:
    xorl %eax, %eax        # %eax = 0
    incl %eax              # %eax = 1, system call _exit ()
    xorl %ebx, %ebx        # %ebx = 0, normal program exit.
    int $0x80              # execute _exit () system call

    ## prints string to stdout
putstring:    .type @function
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %ecx
    xorl %edx, %edx
count_chars:
    movb (%ecx,%edx,$1), %al
    testb %al, %al
    jz done_count_chars
    incl %edx
    jmp count_chars
done_count_chars:
    movl $4, %eax
    xorl %ebx, %ebx
    incl %ebx
    int $0x80
    movl $4, %eax
    leal new_line_char, %ecx
    xorl %edx, %edx
    incl %edx
    int $0x80
    movl %ebp, %esp
    popl %ebp
    ret

```

* The Binutils Package

Binutils stands for binary utilities, and includes a lot of tools useful to programmers, especially during debugging.

I will now address some of these utilities.

** Objdump

Objdump displays information about 1 or more object files. For example, to see information about param-stack, type the following command at shell prompt (be sure working directory contains param-stack);

```
objdump -x param-stack | less
```

Since the information is likely to span more than one screen, output of objdump is piped to the standard input of the pager command 'less'. The option '-x' tells objdump to display numeric information in hexadecimal. Here is the output of the above command;

```
stack-param:      file format elf32-i386
stack-param
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x08048074

Program Header:
  LOAD off      0x00000000 vaddr 0x08048000 paddr 0x08048000
             filesz 0x000000be memsz 0x000000be flags r-x
  LOAD off      0x000000c0 vaddr 0x080490c0 paddr 0x080490c0
             filesz 0x00000001 memsz 0x00000004 flags rw-

Sections:
Idx Name          Size      VMA          LMA          File off  Al
  0 .text          0000004a  08048074   08048074   00000074  2*
             CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data          00000001  080490c0   080490c0   000000c0  2*
             CONTENTS, ALLOC, LOAD, DATA
  2 .bss           00000000  080490c4   080490c4   000000c4  2*
             ALLOC

SYMBOL TABLE:
08048074 l    d  .text  00000000
080490c0 l    d  .data  00000000
080490c4 l    d  .bss   00000000
00000000 l    d  *ABS*  00000000
```

```

00000000 l d *ABS* 00000000
00000000 l d *ABS* 00000000
080490c0 l .data 00000000 new_line_char
08048076 l .text 00000000 again
08048087 l .text 00000000 end_again
0804808e l .text 00000000 putstring
08048096 l .text 00000000 count_chars
080480a0 l .text 00000000 done_count_chars
00000000 F *UND* 00000000
080480be g O *ABS* 00000000 _etext
08048074 g .text 00000000 _start
080490c1 g O *ABS* 00000000 __bss_start
080490c1 g O *ABS* 00000000 _edata
080490c4 g O *ABS* 00000000 _end

```

Notice the Information provided from the program header (ELF files have header information at the beginning of the file giving information to the kernel on how to load the file into memory etc).

ELF files also contain information about the sections (containing section tables). Notice that the .text section contains 0x4a bytes of information, is located 0x74 bytes into the file, and is aligned at a 4 byte boundary ($4 = 2 * 2$), has memory allocated to it (ALLOC), is readable, and contains code (the segment selector cs in this process points to this section (handled by the operating system)).

Information about the symbols is also provided. All this information is used by debuggers and other programming tools to examine binary files.

Objdump can also be used to disassemble binary executables. Typing the following command will disassemble the file to standard output (this does nothing to the actual file, as objdump only reads the file);

```
objdump -d stack-param | less
```

Here is the output of the above command;

```
-----
stack-param:      file format elf32-i386
```

```
Disassembly of section .text:
```

```

08048074 <_start>:
 8048074:      89 e5                movl    %esp,%ebp

08048076 <again>:

```



```

8048076:      83 c4 04          addl   $0x4,%esp
8048079:      8b 04 24          movl   (%esp,1),%eax
804807c:      85 c0             testl  %eax,%eax
804807e:      74 07            je     8048087 <end_
8048080:      e8 09 00 00 00   call  804808e <puts
8048085:      eb ef            jmp   8048076 <agai

08048087 <end_again>:
8048087:      31 c0            xorl   %eax,%eax
8048089:      40              incl   %eax
804808a:      31 db            xorl   %ebx,%ebx
804808c:      cd 80            int   $0x80

0804808e <putstring>:
804808e:      55              pushl  %ebp
804808f:      89 e5            movl   %esp,%ebp
8048091:      8b 4d 08          movl   0x8(%ebp),%ec
8048094:      31 d2            xorl   %edx,%edx

08048096 <count_chars>:
8048096:      8a 04 11          movb   (%ecx,%edx,1)
8048099:      84 c0            testb  %al,%al
804809b:      74 03            je     80480a0 <done
804809d:      42              incl   %edx
804809e:      eb f6            jmp   8048096 <coun

080480a0 <done_count_chars>:
80480a0:      b8 04 00 00 00   movl   $0x4,%eax
80480a5:      31 db            xorl   %ebx,%ebx
80480a7:      43              incl   %ebx
80480a8:      cd 80            int   $0x80
80480aa:      b8 04 00 00 00   movl   $0x4,%eax
80480af:      8d 0d c0 90 04 08 leal   0x80490c0,%ec
80480b5:      31 d2            xorl   %edx,%edx
80480b7:      42              incl   %edx
80480b8:      cd 80            int   $0x80
80480ba:      89 ec            movl   %ebp,%esp
80480bc:      5d              popl   %ebp
80480bd:      c3              ret

```

The ``-d'` tells `objdump` to disassemble sections that are expected to contain code (usually the `.text` section). Using the ``-D'` option will disassemble all sections. `Objdump` was able to give the number of labels in the code because of the information contained in the symbols table.

The first column displays the virtual memory address for each line of code. The second column displays the machine code corresponding to its respective assembler line of code, and finally the code and assembler are contained in the 3rd column.

For more information look in the info documentation system.

** Getting the amount of memory used with size

If you do an `ls -l stack-param' you get the following

```
-rwxrwxr-x    1 robin    robin          932 Sep 15 18:21 stac
```

This tells you that the file is 932 bytes long. However this also contains header tables, section tables, symbol tables etc. amount of memory that this program will use durring run time wil less than this. To find out actual memory use, type the followi

```
size stack-param
```

The above will result in the following output;

```
text      data      bss      dec      hex filename
  74         1         0        75      4b stack-param
```

This tells you that .text occupies 74 bytes, and .data occupies byte, for a total of 75 bytes memory use.

** Getting rid of symbol information with strip

The strip command can be used to get rid of the symbol informat With no options, this command only strips symbols that are not for debugging. With the `--stip-all' option provided, it will s all symbol information, including those used for debugging. recommend not doing this, as this makes the files harder to ana with the standard programming tools. This command is used onl file size is of paramount importance.

* debugging and gdb

Perhaps the most difficult aspect of programming is debugg Quite often the error that caused the program to termi abnormally is not at the line where the program terminated example later on will show this).

Program that exits with SIG_SEGV

```
## stack-param-error.s #####
## Robin Miyagi #####
## http://www.geocities.com/SiliconValley/Ridge/2544/ #####
```

```

## This file shows how one can access command line parame
## via the stack at process start up. This behavior is def
## in the ELF specification.

## Compile Instructions:
## -----
## as --gstabs -o stack-param-error.o stack-param-error.s
## ld -O0 -o stack-param-error stack-param-error.o
#####
.section .data

new_line_char:
    .byte 0x0a
#####
.section .text

.globl _start

.align 4
_start:
movl %esp, %ebp        # store %esp in %ebp
again:
    addl $4, %esp        # %esp ---> next parameter on stack
    leal (%esp), %eax    # move next parameter into %eax
    testl %eax, %eax     # %eax (parameter) == NULL pointer?
    jz end_again        # get out of loop if yes
    call putstring      # output parameter to stdout.
    jmp again           # repeat loop
end_again:
    xorl %eax, %eax     # %eax = 0
    incl %eax           # %eax = 1, system call _exit ()
    xorl %ebx, %ebx     # %ebx = 0, normal program exit.
    int $0x80          # execute _exit () system call

    ## prints string to stdout
putstring:    .type @function
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %ecx
    xorl %edx, %edx
count_chars:
    movb (%ecx,%edx,$1), %al
    testb %al, %al
    jz done_count_chars
    incl %edx
    jmp count_chars
done_count_chars:
    movl $4, %eax
    xorl %ebx, %ebx
    incl %ebx
    int $0x80

```

```
movl $4, %eax
leal new_line_char, %ecx
xorl %edx, %edx
incl %edx
int $0x80
movl %ebp, %esp
popl %ebp
ret
```

Notice that the above program is assembled with the `--gst` option of `as`. This makes `as` put debugging information in our file, such as the original source file, debugging symbols. Using `objdump -x stack-param-error | less` will show you inclusion of debugging symbols.

Now to find out where our error occurred type the following command

```
gdb stack-param-error
```

this will get you to the gdb prompt `(gdb)`;

```
(gdb) run eat my shorts
/home/robin/programming/asm-tut/stack-param-error
eat
my
shorts
Program received SIGSEGV, segmentation fault
count_chars () at stack-param-error.s:47

47      movb (%ecx,%edx,$1), %al
Current language: auto; currently asm
(gdb) q
[~]$ _
```

(gdb will output more than this, I just wanted to highlight what is important).

This tells us that the segmentation fault occurred at line 47 of `stack-param-error.s`. However the problem was caused in line 29. If you look at line 29 of `stack-param.s`, you will see that this reads `movl (%esp), %eax`. This is due to the way intel i386 `leal` handles NULL pointers. `EAX` was never loaded with 0 on a pointer (just some invalid pointer), which caused line 47 to access an area of memory not available to this process (hence segmentation fault). The loop in `_start()` never stopped normally as the condition for breaking out of the loop is `eax` being 0, which never happened.

Debugging is an art that comes with practice. For more information about `gdb`, look in the info pages (e.g. `info gdb`). You can

type `help' at the (gdb) prompt.

The only reason gdb was able to tell you what line number in source code the error occurred is that the debugging symbols source code was included in the output file (recall that we used `--gstabs' option).

Comments and suggestions <penguin@dccnet.com>

=====

You are free to make verbatim copies of this file, providing that the notice is preserved.