CMSC 313 Lecture 08

- Project 2 Questions
- Recap Indexed Addressing Examples
- Some i386 string instructions
- A Bigger Example: Escape Sequence Project

UMBC, CMSC313, Richard Chang <chang@umbc.edu>

Project 2: BSD Checksum Due: Thursday September 30, 2004

Objective

The objective of this programming project is to practice designing your own loops and branching code in assembly language and to gain greater familiarity with the i386 instructions set.

Background

Checksums can be used to detect corrupted files. A file might be corrupted during transmission through a network or because the disk drive where the file is stored is damaged.

The BSD Checksum algorithm uses a 16-bit checksum. Initially, the value of the checksum is zero. For each byte of the file (in sequential order), the checksum is rotated 1 bit to the right and the byte is added to the checksum. The value of the checksum after the last byte of the file has been processed is the checksum of the file.

If the checksum of a file changes, then you know that its contents have been altered. However, it is possible for two different files to have the same checksum (since there are only 64K different values for a 16-bit checksum but many more possible files). So, having the same checksum does not guarantee that the file has not been corrupted. A well-designed checksum algorithm should be able to indicate the most common types of file corruption (e.g., transposed bits, single bits flipped).

Assignment

Write an assembly language program that computes the BSD checksum (algorithm given above) of the `stdin` file. You should output the checksum as a 16-bit binary number to `stdout`. The intention is for you to use Unix input/output redirection:

```
./a.out <ifile >ifile.checksum
```

The value of the checksum can be examined using the `hexdump` command.:

```
hexdump ifile.checksum
hexdump -e '1/2 "%u\n"' ifile.checksum
```

The first `hexdump` command gives the result in hexadecimal. The second `hexdump` command gives the value in decimal. (It is a challenge to alias the second command in Unix.)

Some details:

- Your program must read a block of bytes from the input. You should not read from the input one byte at a time. (It would be terribly inefficient).
- You may assume that when the operating system returns with 0 bytes read that the end of the input file has been reached.
- On the other hand, you may not assume that the end of the file has been reached when the operating system gives you fewer bytes than your block size.

Implementation Notes

- You can check your program using the sum command which prints out the BSD checksum of the file in decimal. (No, you may not call the Unix sum command from your program.)
- Look up the rotate right instruction in the Intel manual to make sure that you are using the correct rotate instruction.
- The BSD checksum algorithm involves adding an 8-bit value to a 16-bit value. Make sure you are doing this correctly.

- You will have two nested loops. The outer loop reads blocks from the input until the end of the file. The inner loop processes one character at a time. Decide ahead of time how the loops are controlled, which value is stored in which register or memory location.
- Record some sample runs of your program using the Unix script command.

Turning in your program

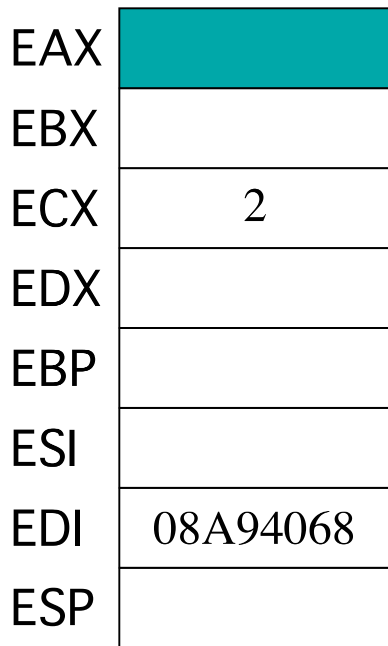
Use the UNIX `submit` command on the GL system to turn in your project. You should submit two files: 1) the assembly language program and 2) the typescript file of sample runs of your program. The class name for submit is `cs313_0101`. The name of the assignment name is `proj2`. The UNIX command to do this should look something like:

```
submit cs313_0101 proj2 checksum.asm typescript
```

Indexed Addressing

- Operands of the form: [ESI + ECX*4 + DISP]
- ESI = Base Register
- ECX = Index Register
- 4 = Scale factor
- DISP = Displacement
- The operand is in memory
- The address of the memory location is
  ESI + ECX*4 + DISP

Base + Index*Scale + Displacement



*4

+

EIP

MOV…

20

Code

Data

08A94068

1734

08A94090

MOV   EAX, [EDI + ECX*4 + 20]

Typical Uses for Indexed Addressing

- **Base + Displacement**
  - access character in a string or field of a record
  - access a local variable in function call stack
- **Index*Scale + Displacement**
  - access items in an array where size of item is 2, 4 or 8 bytes
- **Base + Index + Displacement**
  - access two dimensional array (displacement has address of array)
  - access an array of records (displacement has offset of field in a record)
- **Base + (Index*Scale) + Displacement**
  - access two dimensional array where size of item is 2, 4 or 8 bytes

```
; File: index1.asm
;
; This program demonstrates the use of an indexed addressing mode
; to access array elements.
;
; This program has no I/O. Use the debugger to examine its effects.
;
        SECTION .data                           ; Data section

arr:    dd 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ; ten 32-bit words
base:   equ arr - 4


        SECTION .text                           ; Code section.
        global _start
_start: nop                                     ; Entry point.

        ; Add 5 to each element of the array stored in arr.
        ; Simulate:
        ;
        ;  for (i = 0 ; i < 10 ; i++) {
        ;     arr[i] += 5 ;
        ;  }

init1:  mov     ecx, 0                          ; ecx simulates i
loop1:  cmp     ecx, 10                         ; i < 10 ?
        jge     done1
        add     [ecx*4+arr], dword 5            ; arr[i] += 5
        inc     ecx                             ; i++
        jmp     loop1
done1:

        ; more idiomatic for an assembly language program
init2:  mov     ecx, 9                          ; last array elt's index
loop2:  add     [ecx*4+arr], dword 5
        dec     ecx
        jge     loop2                           ; again if ecx >= 0


        ; another way
init3:  mov     edi, base                       ; base computed by ld
        mov     ecx, 10                         ; for(i=10 ; i>0 ; i--)
loop3:  add     [edi+ecx*4], dword 5
        loop    loop3                           ; loop = dec ecx, jne

alldone:
        mov     ebx, 0                          ; exit code, 0=normal
        mov     eax, 1                          ; Exit.
        int     80H                             ; Call kernel.
```

```
Script started on Fri Sep 19 13:06:02 2003
linux3% nasm -f elf index1.asm
linux3% ld index1.o

linux3% gdb a.out
GNU gdb Red Hat Linux (5.2-2)
...
(gdb) break *init1
Breakpoint 1 at 0x8048081
(gdb) break *init2
Breakpoint 2 at 0x8048099
(gdb) break *init3
Breakpoint 3 at 0x80480ac
(gdb) break * alldone
Breakpoint 4 at 0x80480bf
(gdb) run
Starting program: /afs/umbc.edu/users/c/h/chang/home/asm/a.out

Breakpoint 1, 0x08048081 in init1 ()
(gdb) x/10wd &arr
0x80490cc <arr>:        0       1       2       3
0x80490dc <arr+16>:     4       5       6       7
0x80490ec <arr+32>:     8       9
(gdb) cont
Continuing.

Breakpoint 2, 0x08048099 in init2 ()
(gdb) x/10wd &arr
0x80490cc <arr>:        5       6       7       8
0x80490dc <arr+16>:     9       10      11      12
0x80490ec <arr+32>:     13      14
(gdb) cont
Continuing.

Breakpoint 3, 0x080480ac in init3 ()
(gdb) x/10wd &arr
0x80490cc <arr>:        10      11      12      13
0x80490dc <arr+16>:     14      15      16      17
0x80490ec <arr+32>:     18      19
(gdb) cont
Continuing.

Breakpoint 4, 0x080480bf in alldone ()
(gdb) x/10wd &arr
0x80490cc <arr>:        15      16      17      18
0x80490dc <arr+16>:     19      20      21      22
0x80490ec <arr+32>:     23      24
(gdb) cont
Continuing.

Program exited normally.
(gdb) quit
linux3% exit
exit

Script done on Fri Sep 19 13:07:41 2003
```

```
; File: index2.asm
;
; This program demonstrates the use of an indexed addressing mode
; to access 2 dimensional array elements.
;
; This program has no I/O. Use the debugger to examine its effects.
;
        SECTION .data                   ; Data section

        ; simulates a 2-dim array
twodim:
row1:   dd 00, 01, 02, 03, 04, 05, 06, 07, 08, 09
row2:   dd 10, 11, 12, 13, 14, 15, 16, 17, 18, 19
        dd 20, 21, 22, 23, 24, 25, 26, 27, 28, 29
        dd 30, 31, 32, 33, 34, 35, 36, 37, 38, 39
        dd 40, 41, 42, 43, 44, 45, 46, 47, 48, 49
        dd 50, 51, 52, 53, 54, 55, 56, 57, 58, 59
        dd 60, 61, 62, 63, 64, 65, 66, 67, 68, 69
        dd 70, 71, 72, 73, 74, 75, 76, 77, 78, 79
        dd 80, 81, 82, 83, 84, 85, 86, 87, 88, 89
        dd 90, 91, 92, 93, 94, 95, 96, 97, 98, 99

rowlen: equ row2 - row1

        SECTION .text                   ; Code section.
        global _start
_start: nop                             ; Entry point.

        ; Add 5 to each element of row 7. Simulate:
        ;
        ;  for (i = 0 ; i < 10 ; i++) {
        ;     towdim[7][i] += 5 ;
        ;  }

init1:  mov     ecx, 0                  ; ecx simulates i
        mov     eax, rowlen             ; offset of twodim[7][0]
        mov     edx, 7
        mul     edx                     ; eax := eax * edx
        jc      alldone                 ; 64-bit product is bad

loop1:  cmp     ecx, 10                 ; i < 10 ?
        jge     done1
        add     [eax+4*ecx+twodim], dword 5
        inc     ecx                     ; i++
        jmp     loop1
done1:

alldone:
        mov     ebx, 0                  ; exit code, 0=normal
        mov     eax, 1                  ; Exit.
        int     80H                     ; Call kernel.
```

```
Script started on Fri Sep 19 13:19:22 2003
linux3% nasm -f elf index2.asm
linux3% ld index2.o
linux3%
linux3% gdb a.out
GNU gdb Red Hat Linux (5.2-2)
...
(gdb) break *init1
Breakpoint 1 at 0x8048081
(gdb) break *alldone
Breakpoint 2 at 0x80480a7
(gdb) run
Starting program: /afs/umbc.edu/users/c/h/chang/home/asm/a.out

Breakpoint 1, 0x08048081 in init1 ()
(gdb) x/10wd &twodim
0x80490b4 <twodim>:     0       1       2       3
0x80490c4 <twodim+16>:  4       5       6       7
0x80490d4 <twodim+32>:  8       9
(gdb) x/10wd &twodim+60
0x80491a4 <row2+200>:   60      61      62      63
0x80491b4 <row2+216>:   64      65      66      67
0x80491c4 <row2+232>:   68      69
(gdb)
0x80491cc <row2+240>:   70      71      72      73
0x80491dc <row2+256>:   74      75      76      77
0x80491ec <row2+272>:   78      79
(gdb)
0x80491f4 <row2+280>:   80      81      82      83
0x8049204 <row2+296>:   84      85      86      87
0x8049214 <row2+312>:   88      89
(gdb) cont
Continuing.

Breakpoint 2, 0x080480a7 in done1 ()
(gdb) x/10wd &twodim+60
0x80491a4 <row2+200>:   60      61      62      63
0x80491b4 <row2+216>:   64      65      66      67
0x80491c4 <row2+232>:   68      69
(gdb)
0x80491cc <row2+240>:   75      76      77      78
0x80491dc <row2+256>:   79      80      81      82
0x80491ec <row2+272>:   83      84
(gdb)
0x80491f4 <row2+280>:   80      81      82      83
0x8049204 <row2+296>:   84      85      86      87
0x8049214 <row2+312>:   88      89
(gdb) cont
Continuing.

Program exited normally.
(gdb) quit
linux3% exit
exit

Script done on Fri Sep 19 13:20:35 2003
```

i386 String Instructions

- Special instructions for searching & copying strings
- Assumes that AL holds the data
- Assumes that ECX holds the "count"
- Assumes that ESI and/or EDI point to the string(s)
- Some examples (there are many others):
  - LODS: loads AL with [ESI], then increments or decrements ESI
  - STOS: stores AL in [EDI], then increments or decrements EDI
  - CLD/STD: clears/sets direction flag DF. Makes LODS & STOS auto-inc/dec.
  - LOOP: decrements ECX. Jumps to label if ECX != 0 after decrement.
  - SCAS: compares AL with [EDI], sets status flags, auto-inc/dec EDI.
  - REP: Repeats a string instruction

Project 1: Escape Sequences

Due: Tuesday October 9, 2001 <--- !!!!!! OLD PROJECT !!!!!

Objective

The objectives of the programming assignment are 1) to gain experience writing larger assembly language programs, and 2) to gain familiarity with various branching operations.

Background

String constants in UNIX and in C/C++ are allowed to contain control characters and other hard-to-type characters. The most familiar of these is '\n' for a newline or linefeed character (ASCII code 10). The '\n' is called an escape sequence. For this project, we will consider the following escape sequences:

Sequence	Name	ASCII code
\a	alert(bell)	07
\b	backspace	08
\t	horizontal tab	09
\n	newline	10
\v	vertical tab	11
\f	formfeed	12
\r	carriage return	13
\\	backslash	92

In addition, strings can have octal escape sequences. An octal escape sequence is a '\' followed by one, two or three octal digits. For example, '\a' is equivalent to '\7' and '\\' is equivalent to '\134'. Note that in this scheme, the null character can be represented as '\0'. The octal escape sequence ends at the third octal digit, before the end of the string, or before the first non-octal digit, whichever comes first. For example "abc\439xyz" is equivalent to "abc#9xyz" because the ASCII code for '#' is 43_8 and 9 is not an octal digit.

Assignment

For this project, you will write a program in assembly language which takes a string input by the user, convert the escape sequences in the string as described above and print out the converted string. In addition, your program should be robust enough to handle user input that might include malformed escape sequences. Examples of malformed escape sequences include: a '\' followed by an invalid character, a '\' as the last character of the string and a '\' followed an octal number that exceeds 255_{10}.

All the invalid escape sequences should be reported to the user (i.e., your program should not just quit after detecting the first invalid escape sequence). When the user input has malformed escape sequences, your program should still convert and print out the rest of the string (which might contain some valid escape sequences). In this case, a '\' should be printed at the location of malformed escape sequence. For example, if the user types in "abc \A def \43 ghi \411" your program should have output:

```
Error: unknown escape sequence \A
Error: octal value overflow in \411
Original: abc \A def \43 ghi \411
Convert:  abc \ def # ghi \
```

Turning in your program

Before you submit your program, record some sample runs of your program using the UNIX script command. You should select sample runs that demonstrate the features supported by your program. Picking good test cases is **your responsibility**.

Use the UNIX '`submit`' command on the GL system to turn in your project. You should submit two files: 1) your assembly language program and 2) the typescript file of your sample runs. The class name for submit is '`cs313`' and the project name is '`proj1`'.

Implementation Issues:

1. You should think carefully about how you will keep track of the number of characters you have already processed in the source string. Since you will process more than one character per iteration of the main loop, you will need a consistent way to update the character count and the pointer into the source string.

2. Your program will have numerous branches. You should think about the layout of your program and how to make it more readable. Avoid *spaghetti code*. Related parts of your program should be placed near each other.

3. Do take into account the fact that the output string might be shorter than the input string.

Notes:

Recall that the project policy states that programming projects must be the result of individual effort. *You are not allowed to work together.* Also, your projects will be graded on five criteria: correctness, design, style, documentation and efficiency. So, it is not sufficient to turn in programs that assemble and run. Assembly language programming can be a messy affair --- neatness counts.

Next Time

- Machine Language
- Project 3

References

- Some figures and diagrams from *IA-32 Intel Architecture Software Developer's Manual, Vols 1-3*

  <http://developer.intel.com/design/Pentium4/manuals/>