CS220 - Computer System II
Lab 8

Due: 10/26/2016, 11:59pm

1 Introduction

In this lab, you will implement 32 bit assembly code. You will:

1. Implement Assignment 5 (i.e., reversing a linked list where the links are pointing to middle of the Node structure) in assembly.
2. Measure the difference in performance between your assembly implementation and a reference implementation in C.
3. Interact between C code and assembly code.

2 Getting Started

Download and extract lab8.tar.gz. You will find a folder called Lab8. Inside Lab8, you will find node.h and driver.c. You will implement the assembly code in Lab8. Take a few minutes to examine driver.c. Function reverse_C is the function in C that reverses the linked list starting from head. The function reverse_asm is an external function that you will implement. Also note from signature of reverse_asm that it accepts not only the head of the list, but also the offset from the head of a Node to the link (or ptr field).

3 Implementing reverse_asm

Within Lab7, create `reverse.S`. You will implement reverse_asm function in this file. We will stick to the reversal algorithm used in reverse_C. Open reverse.S and begin your code.

1. First, we will indicate to gcc that we will be coding in intel syntax.

```
1    .intel_syntax noprefix
```

2. We will let the assembler know that the code will reside in the .text segment. We will also declare that reverse_asm will be a function that is accessible in the global context.

```
.text
.global reverse_asm

```

3. We are ready to begin coding our function. Notice that we use 3 variables in reverse_C (headptr, nextptr and new_headptr). Therefore, we could reserve 3 registers for those variables. Remember that eax, ecx and edx registers are saved by the caller. So, we can readily use them for our variables. However, it is usually a good idea to keep eax register free in case we need to move data around. Moreover, eax is the accumulator, so it helps to not commit anything to eax. We could use one of ebx, esi, edi or ebp. (NOT esp!!! Why not?). Let us use ebx. But ebx is not saved by the caller, so it is our responsibility to save ebx, use it, and then restore ebx after we are done. Stack is most suited to save and restore registers. So, the skeleton of our function should look like this:

```
reverse_asm:
  push ebx
  # Now, we can use ebx for headptr, ecx for nextptr, and edx for
  newhead_ptr.
  # Reversal code goes here...
  pop ebx
  ret
```

In assembly, anything that follows '#' is treated as a comment. After this step, the stack looks like Figure 1:

4. We will follow along the code in reverse_C. Note that in intel syntax, destination operand comes before source operand (i.e., mov dest, src):

```
  # headptr = nextptr = new_headptr = NULL
  # We use eax as the scratch register
  mov eax, 0x0
  mov ebx, eax
  mov ecx, eax
  mov edx, eax
```

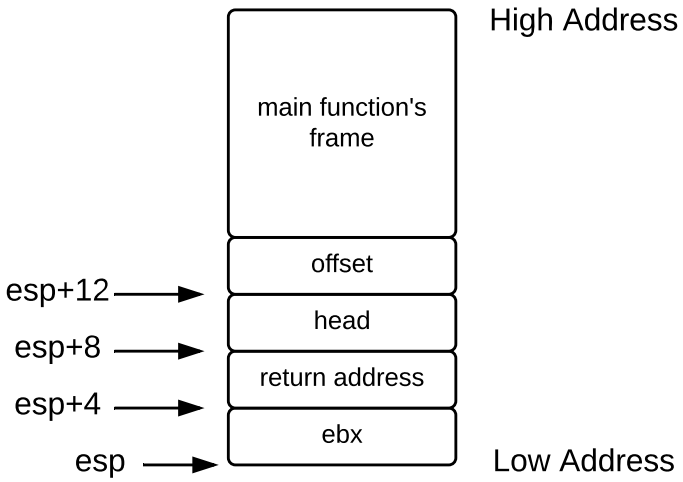

Figure 1: Stack layout after push ebx

5. Next, we handle the corner case where the input is already NULL. Replace offset_of_head_from_current_esp with offset on the stack from current value of esp where head is found. We use the cmp instruction, which is similar to a sub instruction except that it does not update the destination operand with the new value. It simply sets the flags. Particularly, it sets the zero flag if eax == 0. The je instruction jumps to the target if zero flag is set. We also define a label called null_ret that will simply return. Note that eax would already be NULL, so we don't need to set the return value! However, ebx must be restored to prevent corruption.

```
  # This is equivalent to if(head == NULL) return NULL;
2  mov eax, [esp + Offset_of_head_from_current_esp]
   cmp eax, 0
4  je null_ret
# more code comes here...
6 null_ret:
    pop ebx
8   ret
```

6. We set the headptr to the first ptr field of the head node. Replace offset_of_offset_from_current_esp with the offset where the "offset" argument is located. The eax register already contains the headptr, so we add the offset to reach the ptr field.

```
add eax, [esp + offset_of_offset_from_current_esp]
# ebx is headptr
mov ebx, eax

```

7. Now, we implement the while loop. Note that a while loop is nothing but:

```
start:
    if (condition == 0)
        jump end
    execute loop body
    jump start
end:
```

Below, replace headptr, nextptr and new_headptr with appropriate registers.

```
while_start:
  # while (headptr)
  cmp headptr, 0x0
  je while_done

  # nextptr = (void *)(*((unsigned long *)headptr));
  mov nextptr, [headptr]

  # *(unsigned long *)headptr = (unsigned long) new_headptr;
  mov [headptr], new_headptr

  # new_headptr = headptr
  mov new_headptr, headptr

  # headptr = nextptr
```

```
    mov headptr, nextptr

    # next iteration
    jmp while_start

while_done:
    #... code that follows while loop

```

8. After the while loop, the new_headptr points to the ptr field of the new head. We must adjust it to point to the beginning of the new head Node. We will continue to use eax as the scratch register. Replace new_headptr with the right register and offset_of_offset_from_current_esp with offset where the "offset" argument is located.

```
mov eax, new_headptr
sub eax, [esp + offset_of_offset_from_current_esp]
```

9. The eax register already contains the pointer to the new head Node. Also, we have already inserted code to restore ebx and return. So, we are done.

4 Compiling and testing the code

Compile the code using the following command:

```
$ gcc driver.c reverse.S -m32 -std=c89 -g -o driver
$ ./driver
```

You should find a random list with 20 elements printed before and after reversal.

5 reverse_c vs reverse_asm

In this section, we will time the reverse functions in C and asm, and record the findings. Printing is an expensive and time consuming process. So, before you begin, comment the

statements that print the lists.

Functions timeval_subtract and timeval_print are helper functions that will help measure the time taken by a function. In driver.c, within the main function, insert a call to reverse_c, and wrap calls to reverse_asm and reverse_c with the following code to measure the time taken by each variant.

```
int main() {
  struct timeval tvDiff, tvStart, tvEnd;
  ...
      gettimeofday(&tvStart, NULL);
      revhead = reverse_asm(head, offset);
      gettimeofday(&tvEnd, NULL);
      timeval_subtract(&tvDiff, &tvEnd, &tvStart);
      timeval_print("ASM: ", &tvDiff);

      gettimeofday(&tvStart, NULL);
      head = reverse_C(revhead);
      gettimeofday(&tvEnd, NULL);
      timeval_subtract(&tvDiff, &tvEnd, &tvStart);
      timeval_print("C: ", &tvDiff);
      ...
    }
```

Change the value of NUM_NODES to be 10, 100, 1000, 10000, 100000, 1000000, and in each case record the values of time taken by reverse_c and reverse_asm in Lab8.txt. Comment on why one takes more time than the other.

Next, in order to see the effect of optimization, add -O2 flag to gcc during compilation, and repeat the experiments for different values of NUM_NODES. Record your findings in Lab8.txt. Also, comment on how reverse_asm can be made faster.

Obtain the disassembly of `driver` program using:

```
$ objdump -d driver -M intel > driver.disas
```

Count the number of instructions in reverse_c and reverse_asm, and record them in Lab8.txt. Repeat the experiment with -O2 flag added during compilation.

6 Submitting the result

Remove binaries and intermediate files from Lab8. Create a tar.gz of Lab8 folder with only driver.c, node.h, reverse.S and Lab8.txt.:

```
$ tar -cvzf lab8\_submission.tar.gz ./Lab8
```

Submit lab8_submission.tar.gz to Blackboard.