DAT 103
Datamaskiner og operativsystemer (Computers and Operating Systems)

Obligatory Assignment 2 – Words Reversal Assembly Program.

Part I: Bash Script Recap

First let us refresh what we have done in the first part. We wrote three scripts `depunctuate.sh`, `repunctuate.sh` and `words-reverse.sh`. We could then run `words-reverse.sh` either with or without the `--bypass` argument and it (at least the reference solution) would:

1. Make a hash directory, run `depunctuate.sh`, supply it the hash directory as argument and redirect standard input of `words-reverse.sh` to it and pipe the standard output of `depunctuate.sh` through to the standard input of the next program.
    - `depunctuate.sh` would print to standard output the text in lines, either one word per line or the hash of the string containing punctuation characters and for every punctuation string `depunctuate.sh` would populate the hash folder with a file named by the hash of the punctuation string that contained the punctuation string.
2. If `--bypass` was given then we would make the next program `cat` else we wanted to use a program named `words-reverse-ll`
3. Supply the standard output of the previous program into the standard input of `repunctuate.sh` while supplying it the hash folder as a parameter
    - `repunctuate.sh` would use the hash folder to undo what `depunctuate.sh` did

The end effect of running `words-reverse.sh --bypass < LoremIpsum.txt` would be to simply output the text as if nothing had occurred. The objective of this assignment is to actually implement the program `words-reverse-ll` which will perform the desired word reversal before the last step described in our run-through of `words-reverse.sh`.

A denser run-through of what `words-reverse.sh` could look like omitting some details would simply be the line:

```
./depunctuate.sh "$hashdir" <&0 | "$reverser" | ./repunctuate.sh "$hashdir"
```

where the `reverser` variable is set to `cat` or `words-reverse-ll` depending on the `--bypass` parameter.

Part II: Assembly Program

Now you are to implement the part that actually causes the Bash script to perform a reversal. Thanks to the already implemented line- and punctuation splitting, this task is now just a matter of reversing the order of *lines* that the `words-reverse-ll` program receives.

Please read the following before you get started with the tasks.

We have provided the following files:

- `words-reverse-ll.asm`
- `macros.asm`
- `readLine.asm`
- `reverseInputLines.asm`
- `Makefile`

To get started, take a look at the file `words-reverse-ll.asm` (as shown in Listing 1). The file textually includes other files (the `%include "filename"` directive):

- `macros.asm` – contains certain "macros" or helper functionality.
- `readLine.asm` – contains the code implementing the line reading function
- `reverseInputLines.asm` – contains the skeleton of the line reversal function

You are provided with a makefile (the one named `Makefile`) which will build your assembly project as long as you are in the project directory and issue the command `make`. It helps remove some of the tedium of building and linking away.

Listing 1: words-reverse-ll.asm

```
1  %include "macros.asm"
2
3  section .data
4     STDIN equ 0
5     STDOUT equ 1
6     SYS_READ equ 3
7     SYS_WRITE equ 4
8     LINE_SHIFT equ 10
9     buf_size equ 4096
10
11 section .bss
12    input_buffer resb buf_size
13
14 ;  M A I N  E N T R Y  P O I N T
15 ;  ============================
16 section .text
17 global _start
18 _start:
19     ; calling reversal function
20
21     ; sys_exit system call
22
23 %include "readLine.asm"
24
25 %include "reverseInputLines.asm"
```

Next, let us go through some of what one can find in `macros.asm`. The following defines a single-line macro:

```
%define w32FrStck(n) [esp + 4 * (n)]
```

that when used, e.g., in the instruction

```
mov eax, w32FrStck(1)
```

the macro will be replaced and the instruction will become

```
mov eax, [esp + 4 * 1]
```

This instruction will move a 32 bit value (doubleword) from the memory pointed to, offset by 4, by the top of the stack (`esp` contains the stack pointer).
Similarily we have some longer multi-line macros:

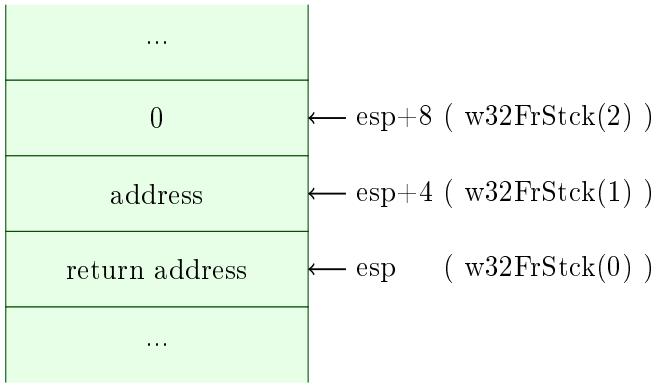
```
%macro call1_1 3
  push dword 0
  push dword %3
  call %2
  add esp, 4
  pop %1
%endmacro
```

That will expand a use of it as in for example `call1_1 edx, routine, address` to

```
push dword 0
push dword address
call routine
add esp, 4
pop edx
```

In this case we push two 32 bit values (doubleword) to the stack, call a routine, adjust the stack pointer, and pop a 32 bit value on the stack to a register.

Keep in mind that the `call` instruction will leave a return address on the stack (the address following the call instruction), so on entry (in `routine`) your stack will look as shown to the right. This return address can be used to transfer control back through a `ret` instruction.



Finally, turn your attention to the multi-line macro:

```
%macro funret1_1 1
  mov w32FrStck(2), %1
  ret
%endmacro
```

which we could use on leaving a function where we have stored a return value say in `eax`, so we would write `funret1_1 eax`, and the stack is as previously shown with a return address at the top of the stack and we have reserved a spot in the stack (`w32FrStck(2)`) to store the return value at.

A Getting started

Open the file `words-reverse-ll.asm` in an editor and navigate to the line after the label that is program entry point `_start:` and in this order write:

1. a call instruction, calling `reverseInputLines`
2. instructions to perform a `sys_exit` system call with an exit code of 0

You can now compile the program by running `make`. Running it will result in a call to `reverseInputLines`, which is at this point not fully implemented. You can test your program by adding a `ret` statement to the top of this function, in which case the program should immediately terminate upon running.

B Read and understand the read line function

In the file `words-reverse-ll.asm` you are provided, you will find that it contains an include directive `%include "readLine.asm"`. This includes a `readLine` function that

- Takes one (4 byte/32 bit/doubleword) parameter: a target address to read bytes to
- Returns the number (in 4 byte/32 bit/doubleword) of characters read (until a newline characters was encountered)

that reads bytes from standard input to a buffer until end of file is reached or a newline character.

This function could have been implemented in C thus:

```
int getLine(char* tgtAddress) {
  int counter=0;
  while (true) {
    char c = getc(stdin);
    if (c==EOF || c=='\n') { return counter; }
    *tgtAddress = c;
    ++tgtAddress;
    ++counter;
  }
}
```

Read through the assembly in `readLine.asm` (shown in Listing 3) and

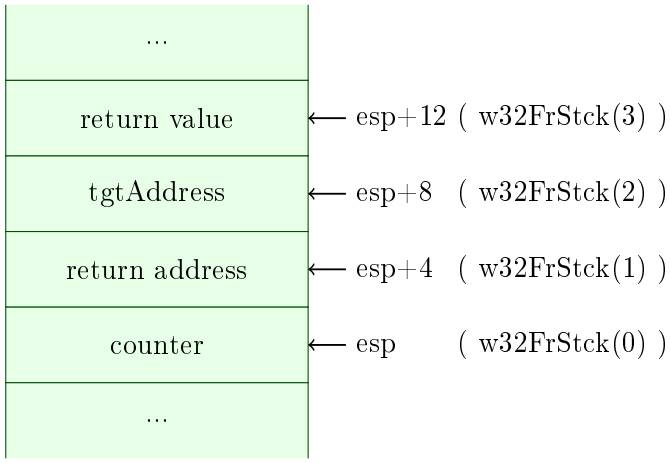
- make sure you understand what is going on
- Note the commenting style; do no worse yourself when you are to implement the line reversal in the next task.

Listing 2: readLine.asm

```
1 ; W O R D  R E A D I N G
2 ; ======================
3 ; int readLine(tgtAddress)
4 ; Make use of the fact that the words (or punctuation-hashes)
5 ; are already split up to lines.
6 ; Result is the number of characters that were in the line.
7 readLine:
8    push dword 0              ; initialize counter
9 readLine_charLoop:
```

```
10      mov eax,SYS_READ
11      mov ebx,STDIN
12      mov ecx, w32FrStck(2)     ; address
13      ; Read one byte/char at a time. Note that this is not efficient.
14      mov edx, 1
15      int 80h
16
17      cmp eax, 0                ; if we're at EOF, nothing is read and we
18      je finished_readLine      ; should finish reading.
19
20      mov ecx, w32FrStck(2)     ; address
21      movzx eax, byte [ecx]     ; the byte we just read
22      cmp eax, byte LINE_SHIFT ; check whether we're done
23      je finished_readLine
24
25      mov ecx, w32FrStck(2)     ; increment address to read to
26      inc ecx
27      mov w32FrStck(2), ecx
28      mov ebx, w32FrStck(0)     ; increment character-in-line counter
29      inc ebx
30      mov w32FrStck(0), ebx
31      jmp readLine_charLoop
32
33  finished_readLine:
34      pop ebx                   ; finished character-counter
35      funret1_1 ebx
```

When calling the `readLine` function you should assume that the 32bit/doubleword starting address pointing to a buffer is pushed to the stack (i.e., function parameters are pushed to the stack). If we then on entry immediately reserve space for the counter variable on the stack we will end up with a stack as shown to the right.



C Implementing a recursive line reversal function in assembly

You will now complete the line reversal program by completing `reverseInputLines.asm`. Base your implementation on the following Python one:

```
#!/usr/bin/env python
import sys
def reverseInputLines():
    l = sys.stdin.readline()[:-1]  # read a line and store the line with newline trimmed away
    if l:                          # in Python empty strings are "falsy"; considered false in if
        reverseInputLines()
        print(l)
reverseInputLines()
```

Note that while you can easily use string variables with arbitrary length, also over recursive calls, in high-level languages like Python, this is not possibly in assembly. In assembly (or C) one must first allocate a suitable amount of memory, as well as later deallocate it again. In real world applications you should usually allocate

on the heap, which is more complicated, but for the sake of our small-scale example we allocate on the stack instead, where this is simply a matter of shifting the stack pointer.

Listing 3: reverseInputLines.asm

```
1  ; R E V E R S I N G
2  ; =================
3  ; void reverse()
4  ; Recursively read in words, until none are found anymore.
5  ; After the recursive calls are done, write out the word again.
6  reverseInputLines:
7
8    call1_1 edx, readLine, input_buffer
9
10   cmp edx, 0           ; If nothing was read, it means the
11   jg there_is_input    ; input is already fully processed, i.e.
12   ret                  ; that this call is finished.
13
14 there_is_input:
15
16   mov eax, esp    ; Original stack pointer
17   sub eax, edx    ; Enough space to store the read string
18
19   mov ebx, 0      ; Complement of round-to-multiples-of-4 bitmask
20   not ebx
21   and eax, ebx    ; Align stack location to 32-bit
22   mov esp, eax    ; Allocate the space on the stack
23
24   mov eax, 0      ; Index into the buffer when copying
25
26   ; TO BE COMPLETED
```

The template `reverseInputLines.asm` already does one half of this allocation/deallocation job: after reading the line, and determining its length to be n characters, it frees up $32 \times \lceil \frac{n}{4} \rceil$ bits of memory on the stack, i.e. $\lceil \frac{n}{4} \rceil$ `dword`s, by shifting `esp` accordingly. (The reason for not just shifting it by n bytes is that this would cause bad alignment.) The rounding is accomplished by a bit mask in lines 19-21, which effectively implements the function $n \mapsto 4 \times \lceil \frac{n}{4} \rceil$.

The first feature you will need to implement is to copy the contents of the string that has been read into this newly allocated stack-memory. Thanks to the 32-bit alignment, you can do that by `moving` one `dword` at a time from the buffer into a register, and writing it into the stack again.

After the string has been copied, you also need to store its length. This is necessary so that you can after the recursive call use the string again for printing, as well as correctly deallocate its memory, which is crucial because the stack is also needed so the previous call levels can correctly proceed.

It is advisable to implement and debug all this first *without* the actual recursion. Check that the string ends up being stored on the stack as intended, that the registers (including `esp`) behave as intended, and that a single word can be printed with the correct length.

When all this works, adding the recursive call in the middle should cause the program to process all the given lines in this way.

Deadline: 31.10.2022, 23:59

Obligatory Submission Part II (due on 31.10.2022, 23:59)

When/Where to submit:

- You will have make your submission in Canvas on/before 31.10.2022, 23:59.

How to submit:

- You can work in a group of at most 3 people. Note that if you work in a group, the group should be formed and registered in Canvas **before the submission**. If you join a group after the submission, a new submission has to be made in order to receive the grade of the assignment.
- Pack the source code you wrote in **a single archive file** called `oblig1-`*studnr*`.tar`, where *studnr* is your student number. If you work in a group, use the group ID instead: `oblig1-group`*grpID*`.tar`.

  For example, `oblig1-4567.tar` if student 4567 submits alone, or `oblig1-group89` for group 89.

  (Please avoid including any spaces, uppercase or non-ASCII characters in the file name. The separator should be *a single hyphen/minus character*, as is good practice in any Unix project.)

  This archive must contain exactly the following:

  1. `words-reverse-ll.asm`, with the call to the reversal function and system exit added
  2. `reverseInputLines.asm`, with the finished function appended, in a well-commented style

  Double-check that your archive conforms to the requirements, by running the sanity checker program that we will provide in by the end of week 42. We may refuse to grade or perhaps even look at assignments that do not succeed the sanity check.