Intro to Linux system calls

This assignment is an introduction to Linux system calls using x86 assembly language on a 64-bit Linux platform.

Your assignment is to implement a bare bones version of the Unix 'cat' utility with a twist. All *cat* does is copy everything it reads from stdin to stdout. Your utility will not recognize any command line arguments, it will simply copy all input from file descriptor 0 (stdin) to file descriptor 1 (stdout). The twist is that you must convert any lower case letters to upper case and upper case letters to lower case BEFORE you write them back out to stdout.

In order to implement your program, you will need 3 system calls: read, write, and exit. The basic algorithm you are to implement is to read a single byte from standard input, test whether the byte is a letter, and if so change its case, then write the, potentially modified, byte back out to standard output. Repeat this process until the return value from read is anything other than 1. When read returns a value other than 1, your program should exit cleanly with a status code of 0.

For this assignment, you are forbidden to use any external library functions such as those from the C library.

Some guidance

You should be able to adapt the hello world example covered in class by adding a read system call, a conditional test on the result of read, a jump if you need to exit, and a loop that makes the whole thing repeat.

All of these instructions as well as how to make system calls are covered to varying degree in the course notes. Here is a short description of the additional instructions that you will need:

Comparing two values is done with the 'cmp' instruction and two operands to compare:

```
cmp rbx, 1  ; compare rbx to one
```

Conditional branching is performed with a conditional jump instruction, generically named 'jcc'. Note that jcc itself is NOT a valid instruction. Rather, the 'cc' is a place holder for a “condition code” which is an abbreviation for the name of the condition that should cause a jump to take place. A detailed list of available conditions may be found under the 'jcc' reference page in volume 2 of the Intel reference manuals. Potentially useful tests for this project will be the “not equal” test, abbreviated 'ne' (alternatively 'nz')

```
jne all_done  ; jump if not equal to label 'all_done'
```

You may also find it useful to test for equality using either 'je' or 'jz' (they are synonymous).

An unconditional jump is performed with the 'jmp' instruction:

```
jmp loop_top  ; jump to the location labeled 'loop_top'
```

To make the required system calls, I recommend reading through the man pages for read and write (‘`man 2 read`’ or ‘`man 2 write`’) in order to understand the required parameters. Make note of the number of required parameters, then map each one to the appropriate register that the kernel expects each argument to be in. You will need one byte of memory allocated somewhere for the kernel to store each character that it reads from stdin. There are many ways to make that happen. Remember that return values from system calls as provided by the kernel differ slightly from return values from system calls as documented in the man pages for C programmers. Specifically for the read system call a return value of 0 indicates that there are no more bytes available (end of file, aka EOF). A positive return value indicates the number of bytes successfully read, and a negative return value indicates some form of error.

Labels

Any instruction may be preceded by a label that associates a name with the instruction so that you may easily reference that instruction as a jump or function call target. You may also label data items to make it easy to reference them as well. The simplest labels are just a sequence of letters, digits, and underscores followed by a colon.

```
label1:
        push qword 1    ; this has label: label1
label2: mov rax, rbx    ; this has label: label2
```

Deliverables

1. Upload your source code to the assignments area on Sakai. Your code must be contained in a single file named exactly **assign3.asm**
2. Your source file must contain, at a minimum, the following information in the form of comments at the beginning of the file:
   a) Your name
   b) The date that you are submitting your program.
   c) Class and assignment number
   d) The command line used to assemble your program into a 64 bit Linux object file
   e) The command line used to link your program into a 64 bit Linux executable file
   f) Any details, such as command line arguments that a user needs to know in order to run your program.

Notes

1. Consult an ASCII table to note any similarities and differences between upper and lower case letters. Make sure that you do not change any other bytes such as numbers and symbols.
2. Do not expect to see output from your program each type you press a key. The kernel buffers your keystrokes to give you a chance to edit your line before the line is passed to your application. When you press the *Enter* key the kernel will make all characters on the newly entered line available to your program which should then be able to read and display them.