Proactive Computer Security
A1: Assembly

Department of Computer Science (DIKU)
University of Copenhagen

C. K. Dalsgaard - *rnq851*

Saturday 1st May, 2021

I have solved all three tasks.

Task 1: Ternary numbers

Structure and register use - I have structured the code in `ternary.asm` into 7 "sections" each with its own label: The instruction in `ternary_convert` sets up the initial result in `rax`, being 0.

At `read_char` a single byte from the adress that the argument of `ternary_convert` points to (`[rdi]`) is moved into register `r8b`. The argument type is `char*`, thus `r8b` is used. Then the pointer is advanced, by incrementing `rdi`, so the next ascii character can be read later on.

The instructions after label `calculate` checks the validity of the character in `r8b`. If it is valid it has value 97, 98, 99 or is a terminating NUL of value 0. If it is NUL we jump to label `done`, which returns. Otherwise the current result in `rax` is multiplied by 3. I.e. if ternary string `"bacacc"` is given, the first char `'b'` represents 1×3^5, so if we multiply `'b'`'s base value with 3 each time we have read a new character in the string we end up multiplying the value with 3 five times, being 3^5. So the base value is added to `rax`, and then multiplied by 3 when the next character has been read.

If an `'a'` is encountered (`r8b` and 97 are equal) we jump back and read the next character. Because `'a'` has base value 0 we do not need to add to the final result, we just multiply by 3. By no means, we could have instruction `add rax, 0`, but it would not change the final result.

If the char is invalid, being less than 97 or greater than 99 we jump to label `ternary_invalid`, where 0 is moved to `rax` and control is returned to the caller of `ternary_convert`.

If an `'b'` is encountered we jump to label `b`, add 1 to `rax` and jump back and reads the next character. The same goes for `'c'` and label `c`, but with value 2.

Challenges - First I implemented the solution by pushing the base values of characters encountered, and when the full string was parsed, I popped them off. I forgot the characters were now received in reversed order, which caused a lot of lost time for me. It was cumbersome, but doable. Then I realised the multiply by 3 trick and implemented it in no time.

Tests - I did test driven development through all three tasks. So I made `ternary_main.c` which uses the function. I printed the interpretation of different ternary numbers of different combinations and length to `stdout`. This was possible by linking my assembly- and `C` file.

Task 2: Grab line

Structure and register use - The entry point and first label `grabline` has the non-volatile registers used set up, both gets initial value 0. `r12` is a counter, keeping track of number of characters read, also used for buffer offset. `r13` is a flag, that gets set to 1 if a newline is read by the extern function `fgetc`. `read_and_handle_char` does what its name implies. `fgetc` is called and returns the value, represented as an `Integer`, of the character read in register `eax`, and -1 if eof. If a newline, eof or 126 characters is read the control flow is changed to label `newline`, `eof` or `done`.

Since we have to write the line of text, character by character and thus byte by byte, to a buffer the register `al` is used. The pointer to the buffer is placed in `rsi`, so writing to it is done using instruction `mov [rsi + r12], al`. We add the offset to the adress, follow the pointer, and moves the byte sized value in `al` here to.

The instructions after `eof` and `done` writes newline and NUL to the buffer, calculates the final number of bytes read and stores it in `rax`, pops the non-volatile registers used and returns.

Challenges - It took some time before I figured out to use `eax` for the return value of `rax`. I reread `man 3 fgetc` and noticed the `int` return value.

Further more I somehow missed to write character 127 to the buffer. I had to move the comparison of the `r12, 126` to the beginning of the loop to solve it.

Tests - I linked my assembly with `main_grabline.c` in which I called `grabline(FILE*, char*)` passing it a text file to read from and a buffer of size 128 to write to. The text files I made was designed to test different aspects of `grabline`. I.e. one contains `"abcd\NUL"` another `"abcd\n\NUL"`, one was empty and another was more than 126 characters long.

Task 3: Reverse file

Structure and register use - At label `reverse_file` I use non-volatile registers `r12, r13, r14`. The first for storing `FILE* inp`, second for `FILE* out` and the third for a line counter, so I do not have to push and pop volatile registers everytime I call a function.

Then the stack is used as the buffer argument for `grabline`, which is the first extern function to be called. I allocate 128 bytes for `grabline` and then I substract another 8 bytes to ensure stack alignment, since three registers initially are pushed on the stack. `grabline`'s return value is stored in register `rax`, if this is 0 there was nothing to read, and we jump to label `done` where the stack space is freed, we pop to the callee save registers and return. If it was not 0 we fall through to label `recur`, which recursively calls `reverse_file`. This call's return value is the number of lines read, so this is added to register `r14`.

At next label `write_line` the call to `fputs(const char* buf, FILE* out)` is set up. The stack pointer `rsp` is passed in `rdi`, and `r13` which contains the `FILE*` to write to is moved to `rsi`. After the call the final result being the number of lines read kept in `r14` is moved to register `rax`, and it fall through to `done`.

Challenges - Task 3 shed the light on the missing nr. 127 character of `grabline`. I ventured back to task 2 to make task 3 work (see).

Tests - I linked my assembly with `main_reverse.c` in which I called `reverse_file(FILE* inp, FILE* out)` passing it a text file to read from and a text file to write to, and then I used `lolcat` to write the text in the `out` file to `stdout`. One file was empty, one too long, one of more than one lines, and so on testing the requirements.