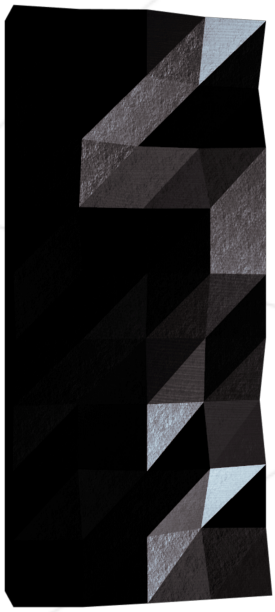

Libasm

Assembly yourself!

Summary: The aim of this project is to become familiar with assembly language.

Version: 5.4

Contents

Chapter I

Introduction

An assembly (or assembler) language, often abbreviated asm, is a low-level programming language for a computer, or other programmable device, in which there is a very strong (but often not one-to-one) correspondence between the language and the architecture's machine code instructions. Each assembly language is specific to a particular computer architecture. In contrast, most high-level programming languages are generally portable across multiple architectures but require interpreting or compiling. Assembly language may also be called symbolic machine code.

Chapter II

Common Instructions

- Your functions should not quit unexpectedly (segmentation fault, bus error, double free, etc.) except in case of undefined behavior. If this happens, your project will be considered non-functional, and you will receive a `0` during the evaluation.
- Your `Makefile` must at least contain the rules `$(NAME)`, `all`, `clean`, `fclean` and `re`. It must recompile/relink only the necessary files.
- To include bonuses in your project, you must add a `bonus` rule to your Makefile, which will incorporate all the various headers, libraries, or functions that are forbidden in the main part of the project. Bonuses must be in different files `_bonus.{c/h}`. The evaluation of the mandatory and bonus parts is done separately.
- We encourage you to create test programs for your project, even though this work **won't have to be submitted and won't be graded**. It will give you a chance to easily test your work and your peers' work. You will find those tests especially useful during your defence. Indeed, during defence, you are free to use your tests and/or the tests of the peer you are evaluating.
- Submit your work to your assigned Git repository. Only the work in the Git repository will be graded. If Deepthought is assigned to grade your work, it will be done after your peer-evaluations. If an error occurs in any section of your work during Deepthought's grading, the evaluation will stop.
- You must write 64-bit assembly. Beware of the "calling convention".
- You can't do inline ASM, you must do '.s' files.
- You must compile your assembly code with `nasm`.
- You must use the Intel syntax, not the AT&T syntax.

`It is forbidden to use the compilation flag: -no-pie.`

Chapter III

Mandatory part

- The library must be called libasm.a.
- You must submit a main function that will test your functions and compile with your library to demonstrate that it is functional.
- You must rewrite the following functions in assembly:
  - ft_strlen (man 3 strlen)
  - ft_strcpy (man 3 strcpy)
  - ft_strcmp (man 3 strcmp)
  - ft_write (man 2 write)
  - ft_read (man 2 read)
  - ft_strdup (man 3 strdup, you can call to malloc)
- You must check for errors during syscalls and handle them properly when needed.
- Your code must set the variable errno properly.
- For that, you are allowed to call the `extern ___error` or `errno_location`.

Chapter IV

Bonus part

You can rewrite these functions in assembly. The linked list functions will use the following structure:

```
typedef struct     s_list
{
  void      *data;
  struct s_list *next;
}           t_list;
```

- `ft_atoi_base` (see Annex V.1)
- `ft_list_push_front` (see Annex V.2)
- `ft_list_size` (see Annex V.3)
- `ft_list_sort` (see Annex V.4)
- `ft_list_remove_if` (see Annex V.5)

The bonus part will only be assessed if the mandatory part is PERFECT. Perfect means the mandatory part has been integrally done and works without malfunctioning. If you have not passed ALL the mandatory requirements, your bonus part will not be evaluated at all.

Chapter V

Submission and peer-evaluation

Submit your assignment to your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Do not hesitate to double-check the names of your folders and files to ensure they are correct.

Annexes

V.1 ft_atoi_base

- Write a function that converts the initial portion of the string pointed to by `str` into an integer representation.
- `str` is in a specific base, given as a second parameter.
- Except for the base rule, the function should behave exactly like **ft_atoi**.
- If an invalid argument is provided, the function should return 0.
  Examples of invalid arguments:
  - The base is empty or has only one character.
  - The base contains duplicate characters.
  - The base contains `+`, `-`, or whitespace characters.
- The function should be prototyped as follows:

```
int        ft_atoi_base(char *str, char *base);
```

V.2 ft_list_push_front

- Create the function `ft_list_push_front`, which adds a new element of type `t_list` to the beginning of the list.
- It should assign `data` to the given argument.
- If necessary, it will update the pointer at the beginning of the list.
- Prototype:

```
void  ft_list_push_front(t_list **begin_list, void *data);
```

V.3 ft_list_size

- Create the function `ft_list_size`, which returns the number of elements in the list.
- Prototype:

```
int ft_list_size(t_list *begin_list);
```

V.4 ft_list_sort

- Create the function `ft_list_sort` which sorts the list's elements in ascending order by comparing two elements and their data using a comparison function.
- Prototype:

```
void ft_list_sort(t_list **begin_list, int (*cmp)());
```

- The function pointed to by `cmp` will be used as:

```
(*cmp)(list_ptr->data, list_other_ptr->data);
```

`cmp could be for instance ft_strcmp.`

V.5 ft_list_remove_if

- Create the function `ft_list_remove_if` which removes from the list all elements whose data, when compared to `data_ref` using `cmp`, causes `cmp` to return 0.
- The data from an element to be erased should be freed using `free_fct`.
- Prototype:

```
void ft_list_remove_if(t_list **begin_list, void *data_ref, int (*cmp)(), void (*free_fct)(void *));
```

- The functions pointed to by `cmp` and `free_fct` will be used as:

```
(*cmp)(list_ptr->data, data_ref);
(*free_fct)(list_ptr->data);
```