Assignment 1 – Sorting in Assembler

Computer Architecture Fall 2017

Part 1 - Programming

Your task is programming a small piece of software only using assembly. You must not link to available libraries or use a higher programming language in order to generate your assembly code. You get access to a couple of implemented functions which will help you to finish the project.

The software has to fulfill the following requirements:

- Pass a filename as command line argument
- Open and read the file into memory
- Sort the numbers stored in the file in ascending order
- Print the sorted list to std out

Specifics

You are free to implement a sorting algorithm of your choice. It is highly advisable to choose an *in situ* algorithm so you don't need to take care of actual memory management.

Assume that the numbers are provided in a file containing **only positive integers between 0 to 32676** (fits in 16 Bit signed integers). Each number is in one row; thus, the numbers are separated by a line feed symbol (ASCII code '0xA'). Remember, this is Unix style, Windows normally uses carriage return followed by a line feed, Mac OS uses only carriage return.

You might provide your code in multiple files. All those files should end with ".asm". The program itself must be able to be compiled with:

```
> as *.asm -o your_sorter.o
> ld your_sorter.o -o your_sorter
```

Ensure that this can be done on the machines in the terminal room. Your program is then called with

```
> ./you_sorter /path/to/file/with/random/numbers.dat
```

The output has to appear be on std out with simply each number in a line:

```
> ./you_sorter /path/to/file/with/random/numbers.dat
12
230
899
>
```

Don't output anything else, neither any greeting message nor debug information.

Hints & Tricks

Snippets

You can download from the project page (link is on the course website and on blackboard) the following code snippets:

`alloc_mem`

This is a function creating some heap space after your program code. As argument, you simply pass the number of required bytes as the first (and only) parameter on the stack. The function returns a pointer to the beginning of the space in %rax.

Note, that this is no actual memory management (Memory is at no point freed, just appended at the end). For our case, this should be sufficient. If you allocating space in a recursive manner, keep that fact in mind. Again, this is the reason you should use an *in situ* algorithm.

`get_file_size`

This function returns the size of a file in bytes in %rax. You will need this function in order reserve enough space to read the entire file into the memory. The parameter, (the file descriptor) is expected to be on the stack. In order to use this function, you have to open the file first in read-only mode which in turn responses with a file descriptor. You should also close the file after reading its content into memory.

`get_number_count`

This function returns the number of integers stored in a given buffer in the memory. For that, you have to pass two arguments on the stack: 1. Address of the buffer, 2. Size of that buffer in bytes. This function allows you in turn to allocate enough space for the actual integers (8 Byte per int for a 64Bit machine).

`parse_number_buffer`

This function scans through a buffer in memory and converts each ASCII representation of a positive integer to an actual integer. The actual integers are stored in a second buffer. Consequently, this function requires three parameters: 1. Address of the raw data buffer, 2. Length of this raw data buffer and 3. Address of the buffer for the actual integers. Here you have to take care that the second buffer is large enough.

`print_number`

This function expects a number on the stack which is then printed to std out. This function is actually register save, meaning no registers are altered by calling this function.

`print_string`

This function prints a null-terminated string to std-out. The address of the string is expected to be on the stack. You can create a zero terminated string for example like this:

```
my_string: .ascii "Hello World\n\0"
```

Creating Random Numbers:

For simple test cases, you can create a file with X random numbers (between 0..32767) quite easily by using the shell:

```
> for i in {1..X}; do echo $RANDOM; done > test.dat
```

Checking the Results

At some point you want to go larger and checking if really everything is in order, you can pipe your program output into a file and then use the sort command:

```
>  ./your_sorter test.dat > output.dat
> sort –nc output.dat
>
```

If something went wrong with sorting, you will see some output, for instance:

```
>  ./your_sorter test.dat > output.dat
> sort –nc output.dat
sort: output.dat:2: disorder: 23179
>
```

Part 2 – Evaluation

After you have finished your sorter, you should evaluate its performance. There are faster and less fast methods for sorting, thus some evaluation should be done. Evaluate your program using the following protocol:

1. Generate test-data sets of the following amount of random numbers: 100; 1,000; 5,000; 10,000; 50,000. For each dataset size, generate 10 different replicates with different random numbers, so we can diminish the influence of luck.
2. Measure the runtime of your program using the command `time`.
3. Display your runtime evaluation in an adequate manner.
4. Count the number of compare-operations you use during sorting. How is the number of compares related to the overall runtime? How is the influence of the ramp-up (everything done before the actual sorting) time depending on the file size? In order to calculate the number of compares, you could simply increment a variable after each compare instruction and output the result at the end. Make sure you make your time measurement without this overhead. The program for submission also must not output any additional information but strictly follow the specifications made above.
5. Calculate the MCIPS (Just invented, that's Million Compare Instruction Per Second) performance of your program for the different input files.

Grading, Submission & Deadlines

Grading

This project will be evaluated with pass/fail marks. In order to pass, you must provide the following items:

Zip-File containing:

- A README file containing your names
- All source code files in order to assemble your program in the subfolder asm/
- Your report as pdf in the subfolder report/
- A table (plain text, tab separated file) containing all time and MCIPS measurements of your evaluation, also in the subfolder report/. The file should have the following format:
  10_time: <time_1> \t <time_2> \t … \t <time_10>
  10_mcips: <mcips_1> \t … \t <mcips_10>
  …
  50000_time: …
  50000_mcips: …

Your source code:

- Must be able to be compiled as described above (no further steps or parameters are allowed) and run on the terminal machines as described
- Has to be commented and you have to explain your steps and decisions in the code
- We highly discourage the use of the Intel syntax, as we won't be able to provide any help to you if you are not using the AT&T syntax.

The report:

- The report (6 pages max) must present the results of your evaluation in a structured, scientific way, following the protocol given above
- Also comment and discuss the results instead of plainly presenting them
- Discuss your motivation for choosing the specific sorting algorithm you implemented

Teamwork:

- **You are allowed and encouraged to work in teams of up to five students.**

Submission

Please upload the zip-File with the contents exactly as described above to blackboard using the to the according SDU assignment tool. We will not accept incomplete zip-Files or files provided by different means than the blackboard system.

Please do only upload one file per team. Make sure that all your team members' names are included in the README file.

Deadline

Upload your zip not later than

Sunday, 12th of November, 23:59:59 Danish Time

Is there something else?

Yes, indeed. As sorting is a quite long-standing problem and the algorithms are not exactly what we would call cutting edge, we still want to give you a motivation to maybe not use the most straight forward algorithm. We hereby announce two challenges:

- Write the fastest sorter fulfilling all requirements
- Write the smallest program, i.e., which program results in the smallest binary

If you can achieve both goals, you will win both prices (Yes, there will be prices). You can also hand in two versions, one optimized for speed, one for size. Put the program you want to be graded on into the folder asm/, your additional program in the folder challenge/. For the challenge, it is highly recommended to NOT use the snippets provided, as they are neither efficient nor compact but easy-to-understand. You can also use MMX or SSE instruction, and use the fact the numbers won't exceed 16 Bit. The only requirement is that the program still is compile-able on the computers in the terminal room and does not use any libraries.