SE 2XA3 (2018/19, Term I) Final project -- lab section L01

Back To Lab Menu | Back To Main Menu | Submissions | Log Out

Final project is an individual project, and thus you are not allowed to cooperate with other people on the solution or design. Exceptions are consultations with the TA's or the instructor.

There is only one task and one deliverable: NASM program `sorthem.asm`. It can be submitted either via the course website, or using `2xa3submit` script from ***moore***. For `2xa3submit` submission please use `2xa3submit AAA xproj1 sorthem.asm` where `AAA` is your student number. You can submit the file as many times as you wish, the latest submission will be used for marking. The submission is opened from 8:30 on Nov 2, 2018 and will close down at 23:00 on Dec. 6, 2018 (there will be no extension possible). If submission is not possible for whatever reason (typically right after the submission closes), email the file as an attachment immediately (needed for the time verification) to Prof. Franek (franek@mcmaster.ca) with an explanation of the problem; you must use your official McMaster email account for that. Include the course code, your lab section, full name, and your student number. Note that if there is no problem with submission (typically the student using a wrong name for the file), you might be assessed a penalty for email submission, depending on the reason you used email.

Before you start you need to download the appropriate supporting files

- asm_io.asm
- asm_io.inc
- cdecl.h
- driver.c (careful, this is a slightly different `driver.c` than used for labs).
- sorthem (this is an executable of the sample solution for ***moore***, if you can (depends on your brower) download it, move to ***moore***, make it executable,and play with it.

NASM program named `sorthem.asm`

The name of your file must be `sorthem.asm` and below is a description of what it should do when executed.

1. `sorthem.asm` expects one command line argument. It must be an unsigned integer bigger or equal to 2 and less or equal to 9. If there are more or fewer command line arguments than one, the program displays an error message and terminates. If the single command line argument does not evaluate to an integer between 2 and 9, an error message is displayed and the program terminates.
2. The program uses an integer array of length 9 to represent a peg. Each item of the array represents the size of the disk at that position on the disk.
   Thus, `[4,3,1,2,0,0,0,0,0]` represents a configuration

```
         oo|oo
          o|o
        ooo|ooo
       oooo|oooo
xxxxxxxxxxxxxxxxxxxxxxx
```

   while `[5,4,3,2,1,0,0,0,0]` represents a configuration

```
          o|o
         oo|oo
        ooo|ooo
       oooo|oooo
      ooooo|ooooo
xxxxxxxxxxxxxxxxxxxxxxx
```

3. Then single command line argument of the value between 2 and 9 represents a number of disks on the peg.
4. The program calls a subprogram `rconf` that creates a random initial configuration. This program is prepared for you in the `driver.c`, so you do not need to worry about. The important thing is to call it properly: first you need to push on the stack the number of disks, then the address of the array representing the peg.
5. The program displays the initial peg configuration as created by `rconf` and pauses and waits for the user to depress a key.
6. The display of the peg configuration is done by a subprogram `showp` that you have to write. The subprogram `showp` expects two parameters on the stack; on top of the stack the address of the array representing the peg, and below the number of disks. Before `showp` returns, it calls `read_char` so the program is paused and waits for the user to depress a key.
7. Then the initial configuration is passed to a subprogram `sorthem` that you have to write. The subprogram `sorthem` expects two parameters on the stack, at the top the address of the peg and below the number of disks.
8. The subprogram `sorthem` actually sorts the disks on the peg in an ~~ascending~~ **descending** order. This is achieved by recursion. It first makes a recursive call to `sorthem` and passes it the peg array starting from the second item, not the first and the number of disks decreased by one. *Hint: if the peg array address is held in* `ebx`*, it will pass to the recursive call the address* `ebx+4` *and if* `ecx` *contains the number of disks, it will pass to the recursive call the number* `ecx-1`.
9. So after the recursive call, the tail of the peg (except the very first item) is sorted. All you need to do is now to find where to put the very first item if the array representing the peg. This is done by swapping the 1st item with the 2nd if the first item is smaller than the 2nd, then swapping the 2nd item with the 3rd, if the 2nd item is smaller that the 3rd, end so on, as long as the swap takes place (see below for example). When the swapping is done, if the peg configuration was modified (i.e. a swap took place) [this aspect is optional], the modified configuration is displayed using `showp`.
10. When done, the final configuration is displayed again using `showp`.
11. After displaying the final configuration, the program terminates.

A sample run with 4 disks (`sorthem 4`) Please note that the sample program does not display the configuration during the swapping process, only after the swapping ends.

```
        initial configuration

       oooo|oooo
         oo|oo
        ooo|ooo
          o|o
XXXXXXXXXXXXXXXXXXXXXXX


       oooo|oooo
         oo|oo
        ooo|ooo
          o|o
XXXXXXXXXXXXXXXXXXXXXXX


         oo|oo
       oooo|oooo
        ooo|ooo
          o|o
XXXXXXXXXXXXXXXXXXXXXXX


         oo|oo
        ooo|ooo
       oooo|oooo
          o|o
XXXXXXXXXXXXXXXXXXXXXXX


          o|o
         oo|oo
        ooo|ooo
       oooo|oooo
XXXXXXXXXXXXXXXXXXXXXXX
```

```
final configuration

          o|o
         oo|oo
        ooo|ooo
       oooo|oooo
XXXXXXXXXXXXXXXXXXXXXXX
```

A sample run with 5 disks (`sorthem 5`) Please note that the sample program does not display the configuration during the swapping process, only after the swappimg ends.

```
 initial configuration

        ooo|ooo
       oooo|oooo
         oo|oo
      ooooo|ooooo
          o|o
XXXXXXXXXXXXXXXXXXXXXXX


        ooo|ooo
       oooo|oooo
         oo|oo
      ooooo|ooooo
          o|o
XXXXXXXXXXXXXXXXXXXXXXX


        ooo|ooo
       oooo|oooo
         oo|oo
      ooooo|ooooo
          o|o
XXXXXXXXXXXXXXXXXXXXXXX


         oo|oo
        ooo|ooo
       oooo|oooo
      ooooo|ooooo
          o|o
XXXXXXXXXXXXXXXXXXXXXXX


         oo|oo
        ooo|ooo
       oooo|oooo
      ooooo|ooooo
          o|o
XXXXXXXXXXXXXXXXXXXXXXX


          o|o
         oo|oo
        ooo|ooo
       oooo|oooo
      ooooo|ooooo
XXXXXXXXXXXXXXXXXXXXXXX


final configuration

          o|o
         oo|oo
        ooo|ooo
       oooo|oooo
      ooooo|ooooo
XXXXXXXXXXXXXXXXXXXXXXX
```

Example of `sorthem` work:
Imagine that the array represening the peg to be sorted as passed to `sorthem` is `1,2,4,3,0,0,0,0,0` and number of disks is `4`. The subprogram `sorthem` will make a recursive call to `sorthem` and passes it peg `2,4,3,0,0,0,0,0` and number of disks 3. After the recursive call completion, the passed array is sorted, i.e. `4,3,2,0,0,0,0,0`, i.e. the whole array is now `1,4,3,2,0,0,0,0,0`. Now `1` must be put in its proper place by swapping: `1` is swapped with `4` getting `4,1,3,2,0,0,0,0,0`, then `1` is swapped with `3` getting `4,3,1,2,0,0,0,0,0`, then `1` is swapped with `2` getting `4,3,2,1,0,0,0,0,0`, and then the swapping ends as `1` is bigger than `0`. Since some swapping took place, the new configuration `4,3,2,1,0,0,0,0,0` is displayed.