IA-32 Assembly Programming — Project Assignment

Simon Kågström
ska@bth.se

Håkan Grahn
Hakan.Grahn@bth.se

September 3, 2013

1 Assembly assignment

Your task is to implement a playable nibbles ("masken") game in IA-32 assembly in a Linux environment.

The examination of this projekt is done by sending in the project, with complete source code and a short report in PDF format, before the examination deadline. The reports and source code should be submitted on It's Learning.

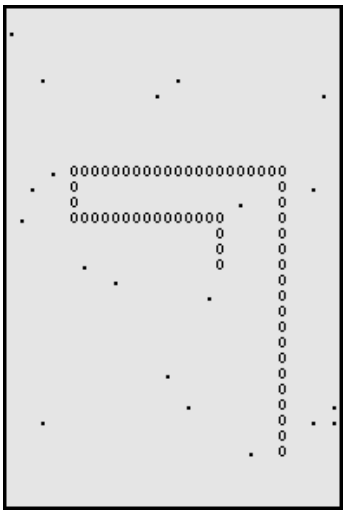

Figure 1: A typical nibbles game.

2 Game requirements

The requirements for the game are listed below.

- The nibbles game must be playable.
  - You must support a configurable number of apples (more than 1)
  - The worm should die if it hits itself.
  - If the worm hits the end of the field (above, below, right or left) it should either:
    * Die, or
    * Wrap around to the other side (i.e. if it exited above the head should reappear at the bottom).
  - Apples should be placed at random positions on the screen and when an apple is eaten, a new apple should reappear. You do not need to handle the situation when the apple appears on the worm (i.e. only the head eats).
- The size of the playfield is not specified, but should be possible to display on the screen.
- The worm should start at the middle of the playfield.
- The game must be implemented *completely* in IA-32 assembly.
- The game must be compilable with `GNU as` or `GNU GCC`, i.e. you should use AT&T-style assembly.
- The source code should be well-commented.
- It should be possible to start the game from either assembly or C. For this, your game must implement the interface specified below.

2.1 Non-requirements

- You do not need to implement:
  - Multiple levels, multiplayer operation etc.
- You are allowed to use tricks, hardcoding etc in the game.

3 Specification

3.1 Prerequisites

- You are supposed to have good programming experience and not be new to C programming.
- You are supposed to have basic knowledge about working in a Unix/Linux environment.
- Operating systems issues and concepts should not be unfamiliar to you.

3.2 Laboratory groups

You are encouraged to work in groups of two. Groups larger than two is not accepted.

Discussion and help between laboratory groups are encouraged. It is normally not a problem, *but* watch out so that you do not cross the border to cheating.

3.3 Lecture support

There are two lectures on IA-32 assembly. In addition to this, you are expected to search for additional information on your own. On It's Learning I have put links to IA-32 Architecture Software Developer's Manuals.

3.4 Provided functionality

You are provided with a number of C functions which you can call from your game implementation. The functions perform initialising, getting keyboard input and printing characters on the screen.

3.4.1 nib_init

`void nib_init(void)`: The *nib_init* function sets up the graphics, initialises the randomiser etc. This function *must* be called before any of the other provided functions are called.

3.4.2 nib_end

`void nib_end(void)`: The *nib_end* function cleans up after the program and exits to the shell. It should be called when the game ends (if the worm dies).

3.4.3 nib_poll_kbd

`int nib_poll_kbd(void)`: The *nib_poll_kbd* function checks for keyboard input non-blockingly, i.e. if the user did not press any key the function just returns. The most important return values are shown below.

-1	There was no keyboard input
'a'..'z'	A letter key was pressed.
258	Down was pressed
259	Up was pressed
260	Left was pressed
261	Right was pressed

3.4.4 nib_put_scr

`void nib_put_scr(int x, int y, int ch)`: The *nib_put_scr* function prints a character on the screen at a specified position.

The character **ch** is just a normal ASCII character code. The coordinates specifies a position from left to right and top to bottom (i.e. 0,0 is the top left corner).

3.5 Useful libc functions

There are also a few useful functions in libc. The most obvious ones are `int rand(void)` and `void usleep(unsigned long usec)`. See the manpages of **rand** and **usleep** for a description of these.

3.6 Files

You are provided with a number of files for the game. These are `helpers.c` which contains the implementation of the functions above, `main.c` which is a small C program which calls your game.

A `Makefile` is also provided which should be easy to customise to your needs. You compile the game by running `make`.

You are expected to implement `nibbles.S` and `start.S` by yourselves.

3.7 Interface

Your game must implement the following interface:

`void start_game(int len, int n_apples)`: Start the nibbles game with an initial worm length of **len** and **n_apples** apples on the screen. Note that this function will be called from C, so that you have to comply with the C calling convention.

`void _start(void)`: The _start symbol, i.e. the entry point of the program. This function should in turn call the start_game function with reasonable parameters.

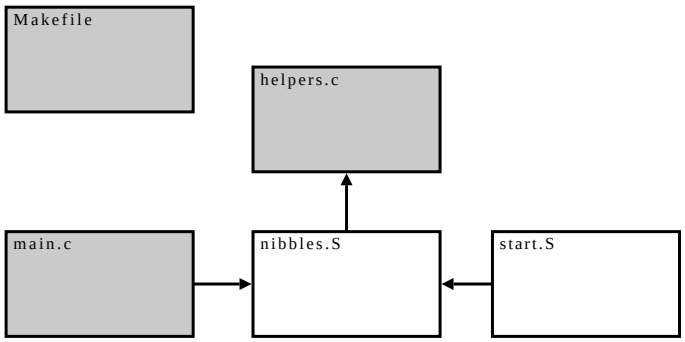

Figure 2: Files in the project

You should also note that these two functions (i.e., global lables in assembly) must be implemented in different files (i.e., `nibbles.S` and `start.S`) since the game should be possible to start from C. This is because the C-program will be linked with `crt0.S` which already contains the **_start**-symbol (and would therefore cause a collision if linked with the file containing your **_start**-symbol).

4 Examination

4.1 Report

You are expected to hand in a report on your nibbles implementation. The report should contain the following things:

1. **Implementation**: A description of your general implementation. I.e., you should describe the datastructures used etc. This part should also contain a printout of the pseudo-code used for your game.
2. **Assembly-related**: How your assembly implementation differs from an (possible) implementation in a higher-level language.
3. **Optimisation**: How you optimised your implementation size. This should describe special instructions used, special data structures etc.
4. **Source code**: A listing of your well-commented source code for `nibbles.S`.

All material (except the code given to you in this assignment) must be produced by the laboratory group alone.

The examiner may contact you within a week, if he needs some oral clarifications on your code or report. In this case, all group members must be present at that oral occasion.

4.2 Grading

In order to get the grade 'A', you must:

1. Use at least three tricks in order to reduce the size of nibbles.o.
2. Explain, how the applied tricks are useful to reduce the size.

In order to get the grade 'C', you must implement:

1. Score.
2. Printouts (e.g. when the worm dies).
3. The worm should grow when eating an apple.

Note: If you do not implement the above requirements, then you will get the grade 'E'.

5 Tips

5.1 General guidelines

- Make the game simple, i.e., don't add any extra features you don't need.
- Make the game working before you start optimising it for the competition.
- It can be a good idea to implement the game in C first before proceeding with the assembly implementation. This will help you find errors in your algorithms etc. You should at least produce pseudo-code before you start implementing the game in assembly.
- **start_game** does not need to return (it should call **nib_end()** when the worm dies and **nib_end()** exits the application).
- Run the game in a terminal with square (or almost square) font, in the size of the playfield. For instance, you can start (assuming your playfield is 80x50) an xterm with

  ```
  xterm -fn 6x9 -geometry 80x50
  ```

  Otherwise the worm will seem to move much faster vertically than horizontally.