Operating Systems
Jacobs University Bremen
Dr. Jürgen Schönwälder

Course: CO20-320202
Date: 2016-11-16
Deadline: 2016-11-22

Problem Sheet #5

Problem 5.1: *letter guessing game - event-driven programming* (2+2+2+2+2 = 10 points)

Your task is to implement a simple letter guessing game. The program outputs a number of random letters that the user has to type into the terminal. The random letters appear on the terminal with a random delay. The user does not see which letters were typed while the game is running. At the end of the game (i.e, the time for guessing letters is over or the user typed a newline, carriage return, or the end-of-file character), the program summarizes the challenge, the user's response, and it shows the number of matching characters.

Below are some example execution. The letters are appearing with a randomized timing in the first line. The letters typed by the user are not visible. The lines following the first line are produced after the game has ended.

```
$ ./letters
Type the following letters: meelmdmcwd
Challenge: meelmdmcwd
Response:  meelmdmcw
Matches:   9 (awesome)
$ ./letters
Type the following letters: sgkwiolnfb
Challenge: sgkwiolnfb
Response:  eeeeeeeeee
Matches:   0 (embarrassing)
```

The default number of letters to guess is 10 (might be changed with the `-n` option) and the default delay between characters is randomly choosen from the interval [0...2000] milliseconds (the upper bound might be changed with the `-d` option).

Your implementation should be entirely event-driven. Feel free to use the `libevent` library if you do not want to implement an event-loop yourself using the `select()` system call. The program will have to handle events from the standard input (the terminal) and you will have to handle timer events (to show additional characters to type). Note that timer events are randomized. You should not make any assumptions in which order events take place.

The programming task can be broken down into smaller activities:

a) Correct implementation of the main event-loop.

b) Correct implementation of an event handler to read characters.

c) Correct implementation of an event handler that produces characters.

d) Proper randomization of characters and timing.

e) Proper implementation of the overall game logic.

To put the terminal into so called raw mode and to reset it afterwards, you can use the following code fragment:

```
static struct termios orig_termios;

static void
reset_terminal_mode()
{
    tcsetattr(0, TCSANOW, &orig_termios);
}

static void
change_terminal_mode()
{
    struct termios new_termios;

    /* take two copies - one for now, one for later */
    tcgetattr(0, &orig_termios);
    memcpy(&new_termios, &orig_termios, sizeof(new_termios));

    /* register cleanup handler, and set the new terminal mode to raw */
    atexit(reset_terminal_mode);
    cfmakeraw(&new_termios);
    tcsetattr(0, TCSANOW, &new_termios);
}
```

Warning: A terminal in raw mode will perform no interpretation of the characters received from the keyboard. In particular, this means no generation of a SIGINT signal if you type Ctrl-C.