Exercise 5

1. Scope and Lifetime of C variables

Study the programs Scope & Lifetime.

Add in the source file global1.c:

- a declaration for an global external char variable named c and
- assign in the `main()` fuction the character ‘a’ to it.

Compile with make, and briefly explain the error that you get and correct the source code.

2. Function Call Frames: CPU Registers + Stack and Text Segments

Let be the following program, similar to the on in the control stack:

```
int g(int n) {
    return n+3;
}

int f(int i, int g(int), int n) {
    return i + g(n);
}

void main() {
    int n;
    n = f(g(2+1), g, 2+1);
    return n;
}
```

For question 1) draw the stack, data and text segments, and the CPU registers (similar to Fig. 4.3 in function call frames), and for the remaining questions 2) â€“ 4) only the stack segments, at the point where the next instruction to be executed is the last semicolon ; of the function calls:

2. `g(2+1)` of `f()`’s first argument
3. `g(n)` of `f()`’s body
4. `f(...)` of `main()`’s body
5. `main()`

assuming that the arguments of a function are evaluated from right to left.

Hints:

- First analyze very carefully f’s prototype (especially its 2nd argument) and its call `f(g(2+1), g(2+1)` in `main()`.
- Secondly draw the figure in similar steps as would happen in a machine: first text and data segments, then stack and heap segments. Note that the register pointers pc, sp, fp point always to “next instruction to be executed”, “next free space on the control stack”, resp. “current function frame”.

3. Pointers as Function Argument + Simplified Stack Segments

Let be `main()`, `swap1()` and `swap2()` as discussed in the Exercise 3 of Series 4. Draw the simplified (e.g. without the pc and fp pointers) stack segments for the function calls

1. `swap1(i, j)`
2. `swap1(&i, &j)` just after the resp. function frame has been built, and just before it is destroyed.
3. Then redo the Exercise 3 of Series 4, but this time for the function `swap2()`.

4. Malloc + Heap Segment

Draw the stack and heap segments when the pc register pointer points to the last semicolon `;` of the following instruction:

```
int* ip;
ip = malloc(sizeof int);
*ip = 3;
```

And why is the casting `(int *)malloc(sizeof int)` not necessary?

5. Calculator: A Stack Machine

Study the flat and modular source code of the calculator discussed in class.

Then for the flat version answer the questions:

2. **About system calls and cache memory.** `getch()` operates as follows: if a character is present in the buffer it takes it, otherwise it calls `getchar()`. Explain why in the former case there is no system call involved, but there is one in the latter case. Explain also why this buffer acts as a kind of cache memory.

3. **About static local variables.** Refactor `getop()` in the simplified case where it handles only operators and non-negative integers (and no floats as in the original version; this simplified code is present as comment in the full version) so that it doesn't need to use `ungetch()`, nor `getch()`. Test your solution on machine. **Hint:** use an internal static variable buf.

And for the modular version:

4. **About static global variables and functions.** Why are the global variables in the files `stack.c` and `ungetch.c` declared as `static`, but not the functions?

5. **About header files.**

    2. Why are the header files `stack.h`, `getop.h` and `ungetch.h` good programming practice?

    3. Why are these files not protected by a conditional directive `#ifndef` (the case for `stack.c` is tricky)?

    4. Write a little program that demonstrate that this protection is not necessary.

Hand in.

Upload your answers on Moodle.