Lab 1: About Memory and Some Useful Tools

Master M1 MOSIG, Université Grenoble Alpes, Grenoble INP

2018

The files for this lab are provided on Moodle (`os_lab1.tar.gz`)[1].

I. Stack and Heap

The *stack* and the *heap* are two distinct memory segments defined by the system for each process so that they can store their data.

- The stack is used to *automatically* allocate memory for the variables defined within functions. Their size is known at compile time and for such variables, memory is automatically reserved when the program enters the function and released when the program leaves the function. It is thus used for temporary storage of information and the use of this information is restricted to the lifespan of the function call.
- The heap is used to store data whose size is unknown at compile time. The memory for this data is to be managed explicitly by the developer, using standard C functions such as `malloc` and `free` (see `man malloc`).

Consider the following code (provided in `ex1.c`) and answer the question:

```
#include <stdio.h>

int min(int a, int b, int c){
  int tmp_min;
  tmp_min = a <= b ? a : b;
  tmp_min = tmp_min <= c ? tmp_min : c;
  return tmp_min;
}

int main(){
  int min_val = min(3, 7, 5);
  printf("The min is: %d\n", min_val);
  exit(0);
}
```

[1]The `tar` command (see `man tar`) can be used to extract the archive. More specifically, use "`tar zxvf myfile.tar.gz`" to extract a `tar.gz` file

Question I.1: *In which memory segment are the variables* `a`*,* `b` *and* `c` *allocated ? When is the memory allocated to them released ? What about the* `tmp_min` *variable ?*

Let us now consider the following code (provided in `ex2.c`):

```
int* vect_sum(int *v1, int *v2, int size){
  int *r, i;
  r = malloc(sizeof(int) * size);
  for(i = 0; i < size; i++){
    r[i] = v1[i] + v2[i];
  }
  return r;
}

int main(){
  int v1[] = {1, 2, 4, 7};
  int v2[] = {3, 4, 9, 2};

  int *p_result = vect_sum(v1, v2, 4);
  /* prints the content of the given vector */
  print_vect(p_result, size);
  exit(0);
}
```

Question I.2: *What value is contained by variable* `r` *after the call to* `malloc()` *inside function* `vect_sum()` *? In which memory segment is this value stored ?*

Question I.3: *What is the exact meaning of the assignment:* `r[i] = v1[i] + v2[i];` *? This assignment results in a (memory) write instruction. In this program, in which memory segment does this write happen ?*

Question I.4: *Write a program that behaves in the same way without using* `malloc()`*. You may need to change the parameters of function* `vect_sum()`*.*

Question I.5: *What is the life cycle of a stack-allocated variable ? of a heap-allocated variable ?*

II. Illegal memory accesses

Correct memory allocation is required for each variable to lie at a distinct place in the memory space. Pointers are very useful, but they also enable to access memory addresses that have not been allocated. If a program tries to read or write at such an address, it may be killed *by the system* with the `SIGSEGV` signal. A memory access that may raise such a signal is called an *illegal memory access*.

Question II.1: *Which lines of this piece of code are illegal memory accesses ? Which one would raise a warning using a "picky" compiler ?*

1. `int *pa = 2;`
2. `*pa = 34;`
3. `int b = 4, *pb = &b;`
4. `*pb = 5;`
5. `int *pc;`
6. `printf("pc is equal to %d\n", pc);`
7. `printf("*pc is equal to %d\n", *pc);`
8. `pc = malloc(sizeof(int));`
9. `*pc = -2;`
10. `pa = pc;`
11. `free(pa);`
12. `pc = -4;`

III. About Makefiles

You may have noticed that a Makefile is provided to you in the archive `os_lab1.tar.gz`. Makefiles are used to automatize the compilation of your code using the `make` utility.

In case you don't know about Makefile, the provided file includes detailed comments that explain how it works. Do not hesitate to look for additional resources on Internet.

To check that you understand how a Makefile works, open the provided Makefile and answer the following questions:

Question III.1: *List the commands that are going to be executed by* `make` *when the following command is executed:*

```
make ex1.run
```

Question III.2: *Same question for the following command:*

```
make ex2.run
```

Start by explaining the variables '`$@`*' and '*`$<`*'.*

Question III.3: *Same question for the following command:*

```
make prog_0.run
```

Start by explaining the use of the symbol '`%`*' in a rule.*

Question III.4: *Same question for the following command:*

```
make all
```

IV. Gdb

`gdb` is a debugging tool. It allows step-by-step execution during which the user can explore the state of the memory (variables, pointers, registers, stack, ...).

Question IV.1: *During your training week you have been provided with a simple gdb tutorial. If you have not been through it yet, it is time to do so. This tutorial is provided to you in the file `gdb-tutorial_EN.c`. Open the file and follow the instructions.*

V. Valgrind

`valgrind` is a tool used to track runtime errors. It simulates the execution of a given executable inside a virtual system, and records any illegal access to the memory as well as other errors.

To use it, simply run:

```
$ valgrind ./my_executable
```

The programs (with a "`prog_`" prefix) given in the archive are all syntactically correct C programs, but they all misbehave at run time.

Question V.1: *Use `valgrind` to find and solve errors in the given `C` files*[2]. *`valgrind` is usually very verbose; write down the `valgrind` errors that helped you and explain their meaning. (If programs are compiled with the `-g` option valgrind is able to provide the source file name and line where the error happened). You may also find the following options useful:*

- *To use `gdb` and `valgrind` together, see http://valgrind.org/docs/manual/manual-core-adv.html#manual-core-adv.gdbserver-simple*
- *--leak-check=yes (to get information about memory that was never freed and are definitely lost);*
- *--show-reachable=yes (to get information about memory that was never freed and are still reachable).*

VI. AddressSanitizer (ASan)

`AddressSanitizer` is another tool that can detect illegal memory accesses. However, it works differently from `valgrind`. `AddressSanitizer` instruments the application source code, and therefore, it requires recompiling the source code.

`AddressSanitizer` is integrated into `gcc` since version 4.8. It is for instance actively used in the development of `chromium` and `firefox` web browsers.

To use `AddressSanitizer`, compile your code with the appropriate flags:

```
$ gcc -g -fsanitize=address my_file.c -o my_exec_file
```

[2] You might want to keep a copy of the initial code of the programs to be able to use them in the next exercise.

Then, you can run your program as usual.

Question VI.1: *Observe the errors in programs with a "`prog_`" prefix using the `AddressSanitizer`. Obviously, you should use the initial version of the codes, that is, without the bug fixes.*

Comparing Valgrind and AddressSanitizer: Both tools can be useful. Each of them may detect errors that the other one is enable to detect. Regarding performance, `AddressSanitizer` is much more efficient than `Valgrind` at the cost of requiring to recompile the application.

Question VI.2: *If you are interested in learning more about `AddressSanitizer`, you can have a look at:*

- *The Github repository:* `https://github.com/google/sanitizers/wiki/AddressSanitizer`
- *The main publication related to this work:* `https://research.google.com/pubs/pub37752.html`

– Bonus –

Bellow this point, the exercises are optional.

VII. Recursive functions

```
int power(int a, int n){
  if( n != 0 )
    return a*power(a, n - 1);
  else
    return 1;
}
```

Question VII.1: *Make a rough estimation of the memory needed to compute* `power(2, 3)`.

Question VII.2: *We provide you with the program* `rec.c`*: try to validate your estimation.*

To do so, you can use the built-in[3] *gcc function* `void * __builtin_frame_address (unsigned int level)` *that returns the address of the function frame (with $level = 0$, returns the frame address of the current function)*[4]

VIII. More of Gdb

Question VIII.1: *We consider again the programs presented in Exercise I. Use* `gdb` *to visualize the state and the evolution of the heap and the stack of the executed program.*

Here are some useful gdb commands:

[3] Built-in functions are functions not defined in the C standard.
[4] See https://gcc.gnu.org/onlinedocs/gcc/Return-Address.html

- *x/nfu addr*

  x command is used to display the memory, starting from addr; n, f, and u are all optional parameters that specify how much memory to display and how to format it.

  - ***n***: *A decimal integer (default 1) that specifies how much memory (counting by units u) to display.*
  - ***f***: *The display format: 's' (null-terminated string), or 'i' (machine instruction). The default is 'x' (hexadecimal) initially.*
  - ***u***: *the unit size. The unit size is any of 'b' (Bytes), 'h' (Halfwords – two bytes), 'w' (Words – four bytes – default), 'g' (Giant words – eight bytes).*

- *info frame*

  to get information about stack frames.

- *p $sp*

  to get the value of the stack pointer.

- *x/5i $pc-6*

  to print 5 instructions 6 words before the current program counter.

Question VIII.2: *Try to use gdb to better understand memory usage for the recursive program introduced in Exercise V.*