Lab 1: About Memory – Answers

Master M1 MOSIG, Univ. Grenoble Alpes (Ensimag / UFR IM2AG)

2025

I. Stack and Heap

Question I.1:

- Variables `a`, `b` and `c` are allocated in the stack.
- Memory is released when the function returns
- The same for `tmp_min`

Question I.2:

- Variable `r` contains the address of the beginning of the allocated memory block.
- Variable `r` is stored in the stack

Question I.3:

- The assignment means that the sum of the values contained at index `i` of arrays `v1` and `v2` is stored at index `i` of array `r`.
- The write is made to the memory space pointed to by `r`, that is, in the heap.

Question I.4:

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

int main(){
  int v1[] = {1, 2, 4, 7};
  int v2[] = {3, 4, 9, 2};
  int v3[4];

  vect_sum(v1, v2, 4, v3);
  /* prints the content of the given vector */
  print_vect(v3, size);
```

```
  exit(0);
}
```

Note: Allocating large data items (for example, a large array or a large data structure) on the stack is not a good practice: indeed, this may lead to a *stack overflow*, because the stack has a relatively limited size.

Question I.5:

- A stack-allocated variable exists only inside the block where it is declared.
- A heap-allocated variable exists as long as it is not freed.

II. Illegal memory accesses

Question II.1:

- Illegal memory accesses:
  - Line 2: Writing to the address pointed by `pa` (2) is not allowed
  - Line 7: Accessing the content of `pc` which is a non-initialized pointer means accessing memory at a *random* address (and hence possibly forbidden).
- Warnings by the compiler:
  - Line 1: initialization makes pointer from integer without a cast.'' (or a variant such as: incompatible integer to pointer conversion initializing 'int *' with an expression of type 'int')
  - Line 6:
    - format '%d' expects argument of type 'int', but argument 2 has type 'int *'
    - 'pc' is used uninitialized
  - Line 12: assignment makes pointer from integer without a cast (or a variant such as: incompatible integer to pointer conversion assigning to 'int *' from 'int')

III. Pointer arithmetic in C

Question III.1:

- The distance is 8 bytes. Beware of the addresses that are displayed as hexa-decimal numbers.
- Pointer `p1` is a pointer of type `unsigned long *`. Adding one to the pointer shifts it by the size of an `unsigned long` (in other words, `sizeof(unsigned long)`).

Question III.2:

The distance can be displayed using the following piece of code:

```
unsigned long *old_p1 = p1;
printf("\n\t\tp1 = %p\n", (void*)p1);
p1++;
printf("after p1++, \tp1 = %p \n", (void*)p1);
printf("distance (in bytes) between p1 and old_p1 = %lu\n",
       (p1 - old_p1)*sizeof(unsigned long));
```

Question III.3:

The distance for `p2` is 80, that is 10 times the size of an `unsigned long`.

Question III.4:

For `p3`, `p4`, and `p5`, the distance is 1 byte. Two main conclusions can be drawn from this result:

- If allowed by the compiler, pointer arithmetic on pointers to `void` behave as with pointers to `char`.
- Unless you are really sure of what you are doing, you should generally perform pointer arithmetic by using type casting to pointer types associated with data types whose size is one byte. Concretely, this means that you should perform pointer arithmetic on pointers defined as `char*` or `uint8_t*`, or otherwise use type casting to such types.

Question III.5:

The compiler raises a warning for the pointer arithmetic operation applied to the pointer of type `void*`. Indeed, by definition there is no size associated to the type `void`. As such, pointer arithmetic on a `void*` is illegal in C (but allowed by some compilers).

Pointer arithmetic using the `void*` type should be avoided (although it is tolerated by some compiler settings).

Question III.6:

The value of `D` is 4. It does not mean that the absolute distance between the two pointers is 4. It means that the distance is 4 times `sizeof(unsigned long)` (`unsigned long*` being the types of the pointers used to run the computation of the distance).

This exercise shows you that pointer arithmetic in C is tricky and must be used with care (and only when really necessary).

IV. About Makefiles

Question IV.1:

- `gcc -c ex1.c -W -Wall -pedantic -g -std=c99`
- `gcc -o ex1.run ex1.o`

Question IV.2:

The variables '$@' and '$<' are automatic variables.

- '$@' is the file name of the target of the rule
- '$<' is the name of the first prerequisite

As such, the commands that are executed are:

- `gcc -c ex2.c -W -Wall -pedantic -g -std=c99`
- `gcc -o ex2.run ex2.o`

Question IV.3:

The use of '%' allows writing pattern rules. The '%' can match any nonempty substring. It enables writing generic rules. Hence the commands that are executed in this case are the one of the pattern rule in the Makefile:

- `gcc -c prog_0.c -W -Wall -pedantic -g -std=c99`
- `gcc -o prog_0.run prog_0.o`

Question IV.4:

The target `all` is used by convention to define a rule that will build all what is needed to make a complete build.

In our case, the target `all` has the list of all the executables as prerequisite. Hence, it will force all executables that have not yet been built or for which a prerequisite has changed since the last build to be built.

Question IV.5:

A *phony* target is one that is not really the name of a file; rather it is just a name for a recipe to be executed when you make an explicit request. See `https://www.gnu.org/software/make/manual/html_node/Phony-Targets.html` for more details.

VI. Valgrind

Question VI.1:

- prog_0.c: Invalid write line 20. The program tries to write to `buf0` after it has been freed.
- prog_1.c: Invalid write line 10. The array is too small because the allocated space is only of `ARRAY_SIZE` bytes instead of `ARRAY_SIZE * sizeof(int)`. Besides, there is also a leak reported by Valgrind because the array is not deallocated (using `free`).
- prog_2.c: The call to `read` returns -1 because the file `/usr/hostname` does not exist (and Valgrind displays a warning about it). As a consequence, line 20 triggers an access to `buf[-1]`, which is outside of the array and corrupts the `r_count` variable (this is

not detected by Valgrind). This results in calling `print_vertical` with an incorrect value for the `len` parameter (too high value), which results in too many loop iterations (going outside of the array), leading to an invalid read at line 12.

- prog_3.c: Invalid write at line 10. The memory for the array has not been allocated.
- prog_4.c: Invalid read at line 16. This comes from the fact that the size of the allocated zone is not sufficient to store the string terminator (`\0`): the requested size should be `strlen(str1) + strlen(str2) + 1`. Besides, once this problem is fixed, Valgrind detects another issue (*"Conditional jump or move depends on uninitialised value(s)"*). This problem comes from the fact that, in the second call to `strncpy`, the last parameter should be `strlen(str2)+1` in order to copy the string terminator character (see `man strncpy`). Also, there is a memory leak, as the dynamically allocated block is never freed.

VII. AddressSanitizer (ASan)

See the previous exercise for a description of the errors in the programs.

VIII. Observing processes virtual address space layout

About the proc file system In a Linux system, the `/proc` directory contains a hierarchy of special files which represent the current state of the kernel.

Among the information that can be found in `/proc`, information about the current state of a process `PID` can be found in the `/proc/PID` directory. For a detailed description of all the information that can be found in `/proc`, we refer you to the man-pages: `man 5 proc`.

Among the virtual files available in `/proc`, we will focus on the information provided by `/proc/PID/maps`. This file describes the state of the virtual address space of a process.

The command `cat` can be used to display the content of a file. Running "`cat /proc/self/maps`" displays information about the virtual address space of the current process.

Question VIII.1:

The process running `cat`.

Question VIII.2:

The answer can be found in the man-pages (`man 5 proc`). More specifically, in the paragraph about `/proc/[pid]/maps`, we can read that the entries are:

- *address*: the address range in the process that the mapping occupies
- *perms*: the set of permissions
- *offset*: the offset into the file/whatever
- *dev*: the underlying device storing the file

- *inode*: the inode on that device (we'll introduce inodes later in the semester, in the lecture about file systems)
- *pathname*: the file that is backing the mapping

Question VIII.3:

The *pseudo-paths* `[stack]` and `[heap]` allow us to identify the stack and the heap in `/proc/[pid]/maps`. The stack grows downwards, thus it is located at the top of the address space. The `[heap]` is then positioned on the opposite side (i.e., at the bottom).

Question VIII.4:

Although the same regions appear, for most of them the starting address differs.

Question VIII.5:

Address space layout randomization (ASLR) is a security technique. It makes it harder for an attacker to locate a specific memory region, and, in particular, to craft an attack based on techniques like buffer overflows (such attacks are typically based on assumptions regarding the precise location of the memory regions and the distance between them).

Bonus question 1: *We could expect to boserve 3 regions corresponding to the 3 main segments of a process address space (in addition to the stack and the heap):*

- *the executable code (`text` segment)*
- *the initialized global variables (`data` segment)*
- *the uninitialized global variables (`bss` segment)*

However, things are more complex (see below).

Bonus question 2: *First we should mention that, depending on how the executable you consider is generated, you might observe 3 or 5 mappings of the executable.*

`In the case you observe 3 mappings`. The first thing to observe is that each region is mapped with different permissions: the first region can be read and executed; the second region can be read; the third region can be read and written. This shows that the previous explanation about the mapping is not correct: the variables in the data segment should be writable.

In fact, the first region really contains the `text` segment. The second section contains read-only information including information related to dynamic linking. The third region (with read-write permission) contains both the `data` segment and the `bss` segment.

`In the case you observe 5 mappings`. The previous explanation is still valid but 2 more mappings have been added. This is because the executable *region described previously has been divided into 3 regions: one that remains executable and still contains the `text` segment, and 2 regions that are read-only because the data they contain do not need to be executable. One of the created region includes information about dynamic linking (among other things), the other includes the `rodata` segment (the read-only data).*

Bonus question 3: *We can see that the virtual address space of the process does not cover the whole set of available addresses. It goes from* `0x00000000` *to something like* `0x7fffffff000`*. It means that a lot of addresses remain available in the address space. The kernel is mapped somewhere in this remaining space (in an address range* above *the one used by the application).*

Bonus question 4: *It simplifies system calls implementation: a system call does not require an address-space switch; passing pointers as arguments to system calls is easier.*

IX. Recursive functions

Question IX.1:

- Running `power(2, 3)` implies 4 calls to the `power` function. Each call to the function requires creating a stack frame that contains the two parameters of the function (that is 2 times 4 bytes for 2 integers). It also contains the return address of the function (8 bytes). Finally, it includes the frame pointer (8 bytes), that is the value of the stack pointer just before the function was called.

  To summarize, each call to the function should use $2 * 4 + 8 + 8 = 24$ bytes
- Running the following code says that 128 bytes are used, that is $4 * 32$. The reason why 32 bytes are used for each frame is that (in the case of Linux/GCC) stack frames are 16-byte aligned (this remark applies both to Intel x86 architecture and to the Arm64 architecture)[1], and so, each call to the function uses 32 bytes instead of 24.

```
void* last_frame;

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

int main() {
  void* first_frame =  __builtin_frame_address (0);
  int pt = power(2, 3);
  printf("The power 3 of 2 is: %d\n", pt);
  printf("amount of memory used: %d\n", first_frame-last_frame);

  exit(0);
}
```

[1] For more information about memory alignment, see the following links:

- General principles: `https://en.wikipedia.org/wiki/Data_structure_alignment`
- Intel x86 ABI: `https://en.wikipedia.org/wiki/X86_calling_conventions`
- Arm64 ABI: `https://johannst.github.io/notes/arch/arm64.html`