CMPUT 379, Assignment 1, Winter 2021

University of Alberta / Department of Computing Science

Instructor: Ioanis Nikolaidis (nikolaidis@ualberta.ca)

(process address space, signals)

Objective

You are asked to program a C function in a single source file with the following synopsis:

```
#define MEM_RW 0
#define MEM_RO 1
#define MEM_NO 2
struct memregion {
        void *from;
        void *to;
        unsigned char mode; /* MEM_RW, or MEM_RO, or MEM_NO */
};

int get_mem_layout (struct memregion *regions, unsigned int size);
```

When `get_mem_layout()` is called, it scans the entire memory area of the calling process and returns in regions the list of all the different regions comprising the process's address space. By region, we mean a *contiguous* area of memory characterized by the same accessibility. Accessibility can have one of three possible values: `MEM_RW` meaning read/write accessibility, `MEM_RO` meaning read-only accessibility, and `MEM_NO` meaning no accessibility at all (the process does not have memory assigned to it in the particular address range).

In essence, your function should try to reference all locations in the address space of the process and find out 1) whether there is readable memory at the respective location or not, and 2) if there is readable memory, whether it is also writable. Note that there exists a unit of memory allocation called a *page* and for a given page size, it suffices to determine the accessibility of a single location within each page. Thus, you don't actually have to scan all locations (for example, successive page base addresses will do).

The function `get_mem_layout()` assumes that the output array regions exists and its size, i.e., the number of entries, is equal to `size`. The function `get_mem_layout()` returns the actual number of memory regions located in the process's address space, even if it is larger than `size`. In the latter case, only the first `size` regions are returned in the array. The regions must be stored in increasing order of their starting addresses. For each `memregion`, `from` represents the starting address of the region, and `to` represents the very last address of the region. Note that `get_mem_layout` should NOT produce any output to `stdout` on its own.

Additionaly, you will write three driver programs (`mem_1.c`, `mem_2.c`, and `mem_3.c`). Each one of the driver programs, first invokes the `get_mem_layout()` function, then prints the entries returned in `memregion` in a humanly readable format, then performs a certain action (different for each action), then invokes `get_mem_layout()` again, and, finally prints the entries of `memregion` (now the result of the second invocation).

You can choose the action of each driver from a set of options. Examples are: a) allocating a massive array with malloc and initializing it, b) `mmap()`-ing a large file, c) invoking a recursive function with a considerable depth of recursion, d) performing a mathematical function using the standard math library, assuming always you compile dynamically loaded executables, etc.

With humanly readable output we mean something mimicking the following format of hexadeciaml address range followed by`RW`, `RO` or `NO` (entries shown here are totally fictitious) :

```
0x00000000-0x0742ffff NO
0x07430000-0x07442fff RW
0x07443000-0x082fffff RO
0x08300000-0x0affffff RW
0x0b000000-0x2fffffff NO
0x30000000-0x3000ffff RO
0x30010000-0xafffffff NO
0xb0000000-0xffffffff RO
```

Notes

There are other ways to determine the memory layout of a process using a variety of tools available under Linux. However, you will have to do it the hard way, i.e., by scanning the memory directly. This is in fact the most reliable way to authoritatively find out what is really there.

Note that you will have to intercept memory reference errors and respond to them in a manner meaningful to the objectives of the exercise. For this, you have to familiarize yourself with UNIX signals and signal handling. They will be covered in the labs.

Because, by default, the `gcc` compiler on the lab machines produces 64 bit executable (which corresponds to a massive address space) you are expected to, instead, produce and run 32 bit executables. This is accomplished with the `-m32` flag.

Deliverables

You should submit your assignment as a single compressed archive file (`zip` or `tar.gz`) containing:

1. A `Makefile`, to compile your test programs (target `all`). The makefile must have a `clean` target as well and specific targets (`mem_1` `mem_2` and `mem_3`) for each driver. `Makefile` should honor (via the definition of `CFLAGS`) in its command line the definition of different page sizes indicated to the C pre-processor as `USER_PAGE_SIZE` (e.g., `CFLAGS=-DUSER_PAGE_SIZE=16384`).

2. The `memlayout.c` and `memlayout.h` files of your implementation where `memlayout.h` includes the prototypes of the two functions and the definitions of `MEM_XX` shown above and the definition of the `memregion` structure. `memlayout.h` also has a definition for a `const unsigned int PAGE_SIZE`. Using pre-processor directives `memlayout.h` should check if `USER_PAGE_SIZE` is set and to use it as `PAGE_SIZE`, and if not defined to set `PAGE_SIZE` to 4096.

3. The three driver programs (`mem_1.c`, `mem_2.c`, and `mem_3.c`)

4. A single `README.txt` file that

    - describes the (three different) actions performed in the three different drivers,
    - describes, for each driver, what differences were found when comparing the memory layout before and after the action taken, and,
    - explains the reason behind the differences found.

Coding Requirements

At this stage of your studies, you are expected to deliver good quality code, which is easy to read and comprehend. A particular facet of quality you need to pay attention to is that your solution **should not cause unnecessary side-effects that modify the very thing it observes**, i.e., it should not result, through its execution, in changes to the address space of the process. Your code must be properly documented at the level of functions as well as at key points in the control flow, and must comply with standard C coding style.

Thursday, January 14th, 2021