CMPUT 379, Assignment 1, Winter 2017

(Address space layout of processes, UNIX 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

struct patmatch {
        unsigned int  location;
        unsigned char mode; /* MEM_RW, or MEM_RO */
};

unsigned int findpattern (unsigned char *pattern, unsigned int patlength,
                          struct patmatch *locations, unsigned int loclength);
```

The array `pattern` of bytes passed as input to `findpattern()` is a sequence of exactly `patlength` bytes which represents a "pattern" (e.g., a fingerprint of a virus) that we would like to check if it is present anywhere in the address space of the process. Note that the elements of the `pattern` array are **not** restricted to ASCII characters – they can be any 8bit values.

Each time `findpattern()` is called, it **has to** scan the entire address space of the calling process and return in the `locations` array all the different locations at which the `pattern` is found. These locations are the **starting** locations of where the pattern was found. Each occurrence of the pattern may be found in locations on a read-only area of the address space (in which case the corresponding `mode` should be set to `MEM_RO`) or on a read-write area (setting `mode` to `MEM_RW`). The array `locations` is allocated by the caller of `findpattern()` and has a length of exactly `loclength` entries. The returned value of `findpattern()` is the count of how many occurrences of `pattern` were found over the entire address space. If the `loclength` is smaller than the number of locations where the pattern was found, then the array `locations` should contain the first (starting from address zero) `loclength` locations where the pattern was found. Regardless of the value `loclength`, the returned value of `findpattern()` is always the total number of locations where the pattern was found.

In essence, your function will try to reference all locations in the address space of the process and, while doing so, it will determine the locations of the occurrences of the given pattern – as well as whether each occurrence is in writable or read-only memory. Note that large regions of the address space are not "mapped", i.e., they do not correspond to any actual memory, and hence nothing can be read from them. You can skip such regions as there is no hope you will find anything there. The unit of memory allocation of the address space is a **page** and you can determine the page size by calling `getpagesize()`. Hence, you could try out to access a location at the beginning of the page (its first byte) and if you fail, to skip to the next page (which could also be mapped or not mapped). This way you can quickly "jump" over unmapped memory and do the actual comparisons only in the memory regions that have mapped memory.

IMPORTANT (a) The address space of the process includes of course the `pattern` array that you pass as input, and therefore you are guaranteed that there will be at least one match. This match is at the location where the `pattern` array is stored. You can use this fact for debugging purposes. (b) We are interested in non-overlapping occurrences of the pattern. For (a grossly simplified) example, assume the pattern is `ababa` and the memory contents are `abababababa`. The result would then be two matches only, one starting at the first location and the second starting at the seventh location.

Notes

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.

Use of `Makefile`s, and their inclusion in the submitted archive, to compile and/or execute your test programs, is *required*. The use of `make` and `Makefile` specifications will also be a requirement in subsequent assignments.

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 `findpattern()` 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, and upon returning it should restore the state of the system to as close as possible (except for returning results of course) to the state of the system as it was immediately before the `findpattern()` invocation.

Deliverables

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

1. At the top directory there should be the files with your implementation of the `findpattern()` function (the name of this file should be `findpattern.c` and a corresponding header declarations file `findpattern.h` with the prototypes of the function and the definitions of `MEM_XX` shown above and the definition of the `patmatch` structure). Note that `findpattern()` should **not** produce any output to `stdout`.
2. A sub-directory `tests` where three example driver programs (`driver[1-3].c`) and any supporting files for those tests will be stored. These driver programs use the function you implemented in this manner: first they perform whatever initializations they need for the purposes of the tests, then they invoke `findpattern()` and print in human-readable form the results returned, then they perform some operation which results in a change of the address space contents, and then `findpattern()` is called again and its results printed in human-readable form, convincing that indeed the pattern is now found in more or different places than originally, or with different modes.
3. The purpose of the three drivers is for you to attempt three **different** ways to introduce or change the location(s) and mode(s) of the pattern. You must **not** prompt for user interaction in order to perform these changes. At least one example should include a demonstration of a change in the `mode` without a change in the `location`.
4. In the `tests` subdirectory, there should be a file `TESTS.txt` that explains the different driver programs, i.e. the different modification approaches.
5. There should be two different `Makefile`s: one on the top directory and one in the `tests` subdirectory. A `make tests` at the top level invokes recursively the `Makefile` of the `tests` subdirectory, compiles and runs the tests, producing their combined human readable output to a file `test_results.txt` in the `tests` subdirectory. Dependencies are expected to exist that will require the re-generation of the `test_results.txt` should the test drivers and any supporting files change. More guidance about `Makefile` targets will be given in the labs.
6. With "human-readable" format we mean that, first you report the returned value from the `findpattern()` invocation, and then the information of the `struct patmatch` entries is printed in 32-bit hexadecimal representation for the `location` and with the symbolic form (MEM_RO, MEM_RW) for the `mode`. Additional text output should help us determine what test is currently running and what is the current stage in the execution of the test.

Sunday, January 8, 2017