TD 1: Virtual memory of a process

1 Compilation

— For information, a very simple way to compile e.g. a `pause.c` program is using
`make pause`
Yes, that's all it takes, nothing more, not even a `Makefile` needs to exist : all predefined rules are enough!

— If you are lazy specifying `pause`, you can create a `Makefile` file :
```
all: pause
```
which tells that the default target is `pause`, and thus a mere `make` just works!

— If you have several programs to compile, you can automatically build them all with :
```
SOURCES=$(wildcard *.c)
PROGS=$(SOURCES:.c=)
all: $(PROGS)
```
which creates the list of `.c` files, then removes the `.c` suffix, and use that as default target. You can then just write `.c` files and run a mere `make` to build them all!

— One can add to `Makefile` :
```
clean:
        rm -f $(PROGS)
```
to be able to run `make clean` to clean the builds.

— Lastly, if you have compilation flags to pass (typically `-g` for the compiler, and `-lz` for the linker), you can add these variables to `Makefile` :
```
CFLAGS = -g
LDLIBS = -g -lz
```

2 Very simple program

— Compile and run the following `pause` program :
```
#include <unistd.h>
int main(void) { pause(); }
```

— Open another terminal, and run `tac /proc/pid/maps` where `pid` is the process identifier of `pause`, you can get it with
`ps -e | grep pause`

— You then get the list of VMAs (Virtual Memory Areas) of your program. The fields are, in order : start-end, rights, offset in the file, device, inode, file. Identify the program, the C library, the dynamic linker, the stack (but forget about the vdso and vvar zones, that's very specific).

— At the end of the slides `cours1.pdf`, there is a summary of the typical VMAs of a program, that's to be learnt by heart!

— Observe that program, linker (`ld.so`) and C library (`libc.so`) are mapped several times, with differing rights, and possibly along a zone not related to a file. These are respectively the constant data, the static data, and the data initialized to zero.

3 Memory position of variables

— Add a global variable `int t[10000];` at the beginning of your program. Recompile, restart, have another look at the `maps` file.
Take care that the pid changes, the simplest is to use `pgrep` :
`tac /proc/$(pgrep pause)/maps`
— Observe that your program indeed now has a zone with data initialized to zero (possibly shared with the heap).
— In the program (before the call to `pause()`), print the address of t :
`printf("t: %p-%p\n",&t[0],&t[10000]);`
Confirm your hypotheses of the previous observation.
— Try to allocate arrays by different ways : as a global variable, as an initialized global variable, as a local variable in `main()`, allocated dynamically with `malloc()`, automatically with `alloca()`. Each time, observe with a `printf()` and the `maps` file which memory zone the array ends up in. Also print the address of `main`.
— With `malloc()`, allocate a small array (less than a KB for Restart the program several times and observe again the `maps`, take care that the pid changes, the simplest is to use `pgrep` :
`tac /proc/$(pgrep pause)/maps`
instance), and a big array (several hundreds of KB). What happens ? The behavior changes around 131 073 bytes. Use `strace ./pause` to check what is happening for the large allocation (instead of a `brk` call).
— Restart the program several times, see that the addresses change ! That's Address Space Layout Randomization (ASLR) to strengthen execution. See which digits change, evaluate how many different possibilities there are. Note that some lower digits do not change, why ?

4 There are not only arrays in life

— Add `#include <dlfcn.h>`, and at the beginning of `main()`,

```
getchar();
void *lib = dlopen("libm.so.6",RTLD_LAZY);
printf("opened libm\n");
```

— Rebuild (adding the `-ldl` option to `LDLIBS` to get access to the `dlopen()` function), restart. Look at the `maps` file. Notice that the `libdl` appeared. That is because we have linked it into the program. Two ways to see which libraries get linked in are :
`ldd ./test`
which gives all the libraries that get loaded (ignore `ld-linux` and `linux-vdso.so` for now), and
`objdump -x ./test | grep NEEDED`
which gives all the libraries that the program itself requests for loading (and which may themselves trigger loading yet more libraries).
— Press enter so that `getchar()` returns, `dlopen()` is then executed, it loads the `libm` library (the mathematical functions library). Look at the `maps` again. Print the addresses of `main`, `printf` and `dlopen`, check where these are.
— Use
`void *sinptr = dlsym(lib, "sin");`
to get the address of the sin function, print it and check where it is.

5 Operating systems are really lazy

— Run `cat /proc/self/smaps` (`self` is a shortcut for "the process that opens the directory", so here the `cat` process). It's the same thing as `maps`, but with some additional information :

Size : the size of the mapping,

Rss : (Resident Set Size) the size currently really mapped,

Pss : (Proportional Set Size) the size currently really mapped divided by the number of processes which map it,

Shared_Clean : the size shared with at least another process,

Shared_Dirty : the size shared with at least another process, and modified,

Private_Clean : the size not shared,

Private_Dirty : the size not shared, and modified.

The `Rss` line is equal to the sum of the 4 `Shared` and `Private` lines.

— Notice that the size currently really mapped is far from being always equal to the size of the mapping.

— Build the following program :

```c
#include <stdio.h>
#include <unistd.h>

int main(void) {
        getchar();
        printf("Hello, %s!\n", getlogin());
        getchar();
}
```

Run it, observe in its `/proc/pid/smaps` file how the figures concerning the `libc-2.28.so` mapping with `r-x` rights change after having pressed enter. You can easily do this by redirecting the output of `cat` to files with `>` and use `diff -U 25` to compare them. Note : if you use `cat`, the figures are below the name of the mapping; if you use `tac`, the figures are above the name of the mapping.

In practice we save a lot of memory by mapping code lazily !

— Create a `dumb` file filled with zeros by running

```
dd < /dev/zero > dumb bs=1k count=16
```

Build the following program :

```c
#include <sys/mman.h>
#include <fcntl.h>
#include <stdlib.h>
    getchar();
    c[4096] = 1;
#include <stdio.h>

int main(void) {
    int fd = open("dumb",O_RDWR);
    if (fd < 0) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    char *c = mmap(0, 16384, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
    printf("%p\n", c);
    getchar();
    printf("%c\n", c[0]);
    getchar();
    printf("%c\n", c[4096]);
    getchar();
```

```
    c[8192] = 97;
    getchar();
}
```

Start it, observe of the figures concerning the mapping of the `dumb` file evolve Try again by running different instances of the program (try different ways of making them progress)
Also look at the content of `dumb` with hexdump.

— Replace `MAP_SHARED` with `MAP_PRIVATE`, try the experiments again.