NOVA SCHOOL OF SCIENCE & TECHNOLOGY
DEPARTAMENTO DE INFORMÁTICA

Memory map and limits

LAB 7

Introduction and Lab objective

The objective of this lab is to introduce the tools and API for memory use, including resource limits and changing a process memory mapping.

Process and computer memory

Remember Lab 1 and the study of Linux's process memory map using the pmap command. You can use the shell process (usually bash command) as the process to study. Use ps to get its PID and then pmap[1] (you can use `pmap -xp . . .` to get the full pathname to the mapped files).

```
fso@linuxfso:~$ ps
    PID TTY          TIME CMD
   1179 pts/1    00:00:00 bash
   1923 pts/1    00:00:00 ps
fso@linuxfso:~$ pmap -x 1179
1179:   bash
Address           Kbytes     RSS   Dirty Mode  Mapping
000056064b7f8000     188     188       0 r---- bash
000056064b827000     772     768       0 r-x-- bash
000056064b8e8000     224     152       0 r---- bash
000056064b920000      16      16      16 r---- bash
000056064b924000      36      36      36 rw--- bash
000056064b92d000      44      28      28 rw---   [ anon ]
000056064d823000    2236    2156    2156 rw---   [ anon ]
00007ffbc6ed1000      64      60       0 r---- libm.so.6
00007ffbc6ee1000     460     248       0 r-x-- libm.so.6
00007ffbc6f54000     360       0       0 r---- libm.so.6
00007ffbc6fae000       4       4       4 r---- libm.so.6
00007ffbc6faf000       4       4       4 rw--- libm.so.6
00007ffbc6fb0000      28      28       0 r---- libnss_systemd.so.2
00007ffbc6fb7000     208     208       0 r-x-- libnss_systemd.so.2
[. . .]
00007ffbc7571000      40      36       0 r---- ld-linux-x86-64.so.2
00007ffbc757b000       8       8       8 r---- ld-linux-x86-64.so.2
00007ffbc757d000       8       8       8 rw--- ld-linux-x86-64.so.2
00007ffc7f9f7000     132      56      56 rw---   [ stack ]
00007ffc7fae2000      16       0       0 r----   [ anon ]
00007ffc7fae6000       8       4       0 r-x--   [ anon ]
---------------- ------- ------- -------
total kB           10300    6700    2428
```

Now you can better understand the mapped libraries in the process memory space, and the dynamic linker `ld-linux-x86-64.so` in the process address space. Some of those areas can possibly be shared with other processes. Can you identify the candidates? Also, some pages, are not in physical memory (look at Kbytes vs RSS). So, from approximately 42 GB of address space, this process uses aprox. 10MB virtual memory, but only aprox. 6MB are in physical memory and, from these, several MB can be shared with other processes. Should be no surprise that, if you add all the memory used by all the processes,

[1] On MacOS use vmmap command.

you reach a memory usage much above your computer real memory. You can check the real memory and extra swap space with the following command[2]:

```
fso@linuxfso:~$ free -h
               total        used        free      shared  buff/cache   available
Mem:           1.4Gi       576Mi       635Mi       1.9Mi       391Mi       886Mi
Swap:          974Mi          0B       974Mi
```

Gi - o i potencia de 2 em vez de potencia de 10

Resource limits

Besides the OS Vitual Memory limit, each process can have stricter restrictions when executing, for memory and for other resources. Those are defined at user login for its first process and then, will be inherited by its descendants. In a terminal, check the limits for your processes by running the command:

```
fso@linuxfso:~$ ulimit -a
real-time non-blocking time  (microseconds, -R) unlimited
core file size              (blocks, -c) 0
data seg size               (kbytes, -d) unlimited
scheduling priority                 (-e) 0
file size                   (blocks, -f) unlimited
pending signals                     (-i) 5749
max locked memory           (kbytes, -l) 187288
max memory size             (kbytes, -m) unlimited
open files                          (-n) 1024
pipe size                (512 bytes, -p) 8
POSIX message queues         (bytes, -q) 819200
real-time priority                  (-r) 0
stack size                  (kbytes, -s) 8192
cpu time                   (seconds, -t) unlimited
max user processes                  (-u) 5749
virtual memory              (kbytes, -v) unlimited
file locks                          (-x) unlimited
```

Those are the soft limits. Some can be increased until its hard limit. Add -H to the previous command to see those. In this example, there are very few limits. Examples: 8MB for stack and 1024 open files; also, you can have a maximum of 5749 processes/threads. Run the provided memtest.c to query and print the resource limits as seen by a process:

```
fso@linuxfso:~$ memtest
addres space limit: soft limit=ffffffffffffffff, hard limit=ffffffffffffffff
data limit:         soft limit=ffffffffffffffff, hard limit=ffffffffffffffff
stack limit:        soft limit=          800000, hard limit=ffffffffffffffff
```

Change the program to create an array of 9Mbytes in the doit function (it will be allocated in the stack) and add an infinite loop at the end. Try to execute. Explain why you can't call your function in your program.

Try changing the following limits in your terminal, and run your program again:

```
fso@linuxfso:~$ ulimit -t 3
fso@linuxfso:~$ ulimit -d 10000
fso@linuxfso:~$ ulimit -s 10000
fso@linuxfso:~$ memtest
addres space limit: soft limit=ffffffffffffffff, hard limit=ffffffffffffffff
data limit:         soft limit=          9c4000, hard limit=          9c4000
stack limit:        soft limit=          9c4000, hard limit=          9c4000
Killed
```

Explain why it runs, the new values printed and why the infinite loop is terminated after some time.

[2] On MacOS you can use GUI Monitor application or sysctl -a|grep -e swapusage -e memsize

Testing mmap

Read the manual page and bibliography about mmap. Try the program `testing_mmap.c`. That uses mmap to map a file to its virtual memory and reads that file by reading that memory.

Sharing memory with mmap

Check the program `sharemem.c` where an anonymous memory area, created with mmap, is shared by two processes (father and son). In this example both processes read and write an array shared by both. Confirm that the father process reads values written by its child.

Readers and Writers processes

In the provided *stocks* subdirectory, there is a demo of the readers/writer pattern. Imagine a stockbroker where one program receives all the stocks transactions that several other users/programs must see. This program is in `stocks.c`, and the data to be shared is simulated by reading each line of the *stock.trace* file at some time intervals. All this data is published on a "board", for all the other programs to read (the *clients*), using a shared memory array. Each position of the array includes a sequence number, company code, number of shares traded and its price. The implemented `client.c` program just reads the content of that array and prints to standard output. A simple check is made to the consistency of that the information. That is, using the sequence numbers, the *client* verifies if the board includes all the last 10 transactions, without missing or repeated entries. These errors can occur if a client reads the array in the middle of the update by the *stocks* program.

You need to add the needed shared memory and concurrency control implementations. Remember that you must use 'named shared memory' and 'named semaphores', as the is no father/child relations, processes can be executed at any time and there can be multiple clients running at the same time. Also, since you don't need a file for *mmap* your shared memory, use `shm_open`. Note that, in this case, you need to use `ftruncate` after `shm_open`, to extend the shared memory to the pretended size.

Bibliography

[1] Sections about Concurrency of the recommended book, "Operating Systems: Three Easy Pieces Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau"

http://pages.cs.wisc.edu/~remzi/OSTEP/threads-cv.pdf

http://pages.cs.wisc.edu/~remzi/OSTEP/threads-sema.pdf

[2] Slides from theoretical classes (in CLIP) particularly: Aula 10 and Aula 13