OPERATING SYSTEM

TD 3: Virtual Memory

1 Organization of the virtual process space

(1.1) Deduce from `/proc/self/maps` how the different memory regions of processes are organized in virtual memory under Linux. Why are the stack and the heap positioned like this?

(1.2) Where is the kernel memory located? Why is it a part of the virtual user space of the process while there is no access?

(1.3) What variations are observed from one process to another, even if they are launched exactly in the same way? Why is it useful to make the mapping vary in user space? Why is it useful to not to make the kernel mapping vary?

(1.4) Make some private and shared `mmap`, and find the corresponding entries in `/proc/self/maps`.

(1.5) Write a program that invoke a lot of of small `malloc` (a few kilobytes or tens of kilobytes) and explain the evolution of the heap(s). Now make big `malloc` (megabyte) and explain too. Deduce the behavior of the system with respect to a possible heap overflow.

(1.6) Write a recursive program that overflow the stack. To know when he failed, you can regularly display `/proc/self/maps`. You can also try to catch the signal `SIGSEGV` to display the exact address where the overflow occurred (this will probably require `sigaltstack`). Why can the system not manage the stack overflows like the heap overflows?

(1.7) For this question, use a 64bit machine for interesting results. Use `uname -m` to find out if this is the case.

Determine the maximum size that can be mapped in virtual memory at one time with `mmap`. You will use flags `MAP_ANON|MAP_PRIVATE` and descriptor `-1` to get an anonymous mapping (without associated file). Try again with `MAP_NORESERVE` and explain.

Make a large `mmap` loop until the system refuses to allocate new areas. Deduce an approximation of the maximum size of the address space of the process. Compare this size to the concept of 64bit architecture, especially by looking at `/proc/cpuinfo`.

(1.8) Try to map a page at the address $0x1000$ (second virtual page). What happens? Why?

Try to map a page at the NULL address. How to do that, and what happens? What is the goal of all these behaviors?

2 Virtual Memory Manipulation

(2.1) Create a 1MB file. Write a program mapping (`mmap`) 3× this file publicly (`MAP_SHARED`) in the same process. Modify the file content in two mappings, then read the modified content in the another one.

(2.2) Replace one mapping by a private mapping (`MAP_PRIVATE`) and explain what happens. Find a way to observe that virtual memory management is done at the page granularity.

(2.3) Modify your program to create only one 1MB public mapping (and none private). Add a command line argument to finish after one of the intermediate steps:

- ✓ before `mmap`,
- ✓ after `mmap`,
- ✓ after reading a byte, a page, or all the mapping,
- ✓ after writing a byte, a page, or all the mapping,
- ✓ until the end of the program.

Use `/usr/bin/time` to measure the page faults of your process depending on the step you are stopping. Explain the results.

3 Page table

Consider a virtual memory with a MMU. The addressing capability of the processor is 32 bits. The page size is 4kb.

(3.1) How many pages in a virtual space? What is the size of the virtual addresses?

(3.2) What is the size of the page table if it is represented linearly and if each entry contains the number of a 48-bit physical frame, 3-bit for the rights, one `VALID` bit, one `DIRTY` bit and 11 `AGE` bits?

(3.3) Which information allows to know if a page access is valid? How to know which pages will be used in the future? What information allows to know if a page is present in memory? What is the size of the physical addresses? What do you think about?

(3.4) Is it possible to store the page table in the MMU? Is it possible to store it in physical memory? What happens if we have 200 processes?

We now consider a 3-level page table. Levels 1 and 2 contain 4kb tables filled with pointers (32bits) to the next levels. Level 3 contains tables of 4kb containing the entries of the above-mentioned linear table.

(3.5) Explain in a diagram how to translate a virtual address into a physical address by browsing this table. In particular, explain how each bit of the virtual address is used.

(3.6) Compare the occupation of this table with the one of the above-mentioned linear table when a process uses 1kb, 100kb, 10Mb, 1Gb then all the available space. What about a table covering $1000\times$ 1kb distributed throughout the virtual space?

4 Entry states of the page table

(4.1) Describe the state of the rights, VALID, and DIRTY bits of an entry of the page table in the following cases:

- ✓ Invalid address
- ✓ Valid address pointing to read-only page, present in memory
- ✓ Valid address pointing to write-only page, present in memory and recently modified
- ✓ Valid address pointing to read-write page, but currently in copy-on-write
- ✓ Valid address pointing to missing page in memory because swapped on disk

(4.2) In which case will the system have to load a page from disk into physical memory? In which context is the operation performed? Where are stored the required informations to locate the page on the disk?

5 Translation Lookaside Buffer Translation

Our processor have now equipped a TLB but no MMU.

(5.1) What information the TLB needs? What information the TLB does not need? Why only a part of the virtual memory management informations are currently placed in the TLB? Compare this with a MMU.

What happen when the TLB does not have the translation of a requested address? Make the diffrence between valid, invalid, and copy-on-write acceses. Again, compare with the MMU case.

(5.2) What instructions should the processor know to use the TLB?

(5.3) Consider the case of a multi-core processor. How many TLBs can it or should it have? How do they interact?