INFO3203 - ARCHITECTURE & SYSTÈME 2023-2024

TP 2

MAKING MEMORY MANAGEMENT THREAD SAFE

In this TP, you will build on the memory management system you implemented in TP1 to make it thread safe.

NOTE: You can both build on the solution you implemented yourself and submitted for TP1 or use the solution provided here: TP1 solution

CONTEXT

Memory management in operating systems has two components.

1. The first component is a physical memory allocator for the kernel, so that the kernel can allocate memory and later free it.
2. The second component is virtual memory, which maps the virtual addresses used by kernel and user software to addresses in physical memory.

A number of rules are applied when working with multiple processes:

1. Pages are allocated to processes. This means that each page has a process that owns it and it's the sole user of the page.
2. Consequently, each process has its own virtual memory space, i.e., virtual addresses are valid only within a process and cannot be accessed by other processes.
3. Allocation (`vmap_malloc()`) and deallocation (`vmap_free()`) operations are thread safe. This means that the system has to guarantee that if two threads (obviously belonging to the same process) perform a `vmap_malloc()` they will not be handed the same memory space.

THREAD SAFE IMPLEMENTATION

The goal of this TP is integrate the memory management system developed in TP1 to comply with the rules specified in the previous section. Ultimately, the operations performed by your memory management system will not change, i.e., you will still access memory using the same three operations:

- `void *vmap_malloc(struct VMap *vmap, int size)`
- `void vmap_free(struct VMap *vmap, void *data)`
- `int vmap_copy_to_memory(struct VMap *vmap, void *mem, void *data, int size)` and `int vmap_copy_from_memory(struct VMap *vmap, void *data, void *mem, int size)`

TIPS

While the are a large number of ways the requirements specified above could be met, here are a number of tips that might reveal useful in implementing your solution:

1. The page ownership system requires that each page is assigned to one and one only process. My suggestion, though, is not to modify the physical memory component almost at all (with the exception of the swap, see below).
2. This means that all page ownership can be handled in the virtual memory map component. Ultimately, this means to duplicate for each process the structures used to control virtual memory. How to know which process is invoking a function? You should identify each thread with a PID-like variable to make sure each thread can only access to his virtual page. You should also handle errors if a thread is trying to access the wrong virtual page.
3. If all the above makes sense, making `vmap_malloc()` and `vmap_free()` thread safe should be a matter of smartly using mutexes.
4. What about the swap? As we implemented the swap using a single resource, i.e., a file, you will need to properly protect it from concurrent access.

SUBMISSION

Submit all the developed code, including the memory management system parts, in a single archive using the submission page by 05/05/2024 at 19h00.

NOTE: Make sure to simplify the life of the lab instructors by including as much documentation (e.g., README with explicit test command, comments in the code) as possible. Do not forget to include a Makefile.

Modifié le: vendredi 19 avril 2024, 08:04