INFO3203 - ARCHITECTURE & SYSTÈME 2023-2024

TP 1

MEMORY MANAGEMENT (PART 2)

In this TP, you will continue to study how (and develop) the memory management of an operating system works. Starting from the code base developed in TD 5, you will build your own memory management system 😀.

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. You have implemented a number of functions that replicate this first component during TD 5.

2. The second component is virtual memory, which maps the virtual addresses used by kernel and user software to addresses in physical memory. The x86 hardware's memory management unit (MMU) performs the mapping when instructions use memory, consulting a set of page tables.

As you probably imagined, the goal of this TP is to complete the job and implement the second component! At the end of the TP, you should be able to run simple applications on top of your memory management system.

The pagination implementation (Mostly TD 5)

In the last TD, you have implemented a number of functions that manage pagination between a "physical" memory and the swap. The functions you implemented, required you to: (1) Create a "physical" memory; (2) Allocate pages; (3) Move them to "swap" and read them back.

To make sure we are all aligned on the implementation of these, you can find my own solution at this link: Allocator

NOTE: You were not instructed on which data structure to use to store the information related to allocated / free pages. The solution provided uses the possibly "simplest" solution – a linked list. Unless instructed so, performance is not the main goal of this course. Obviously, if your implementation uses more efficient data structures that is good! For the rest of this document, I will document functions using structure names based on my solution. You are free to replace them with your own.

To complete the pagination implementation, there remain X tasks to complete. In particular, you should write the following functions:

- `struct PageInfo *page_lookup(struct MMap *mmap, int key)`: to get a page based in its unique key. Remember that the page might be in the allocated list or in the swap (or not exist for what that matters)
- `void page_remove(struct MMap *mmap, int key)`: to delete a page based on its unique.
- `struct PageInfo *page_create(struct MMap *mmap, int *moved_to_swap)`: to create a new page. Note that `moved_to_swap` is used to signal whether a page had to be moved to swap. If that is the case, the content of is changed to the key of the page moved to swap.

Your page allocator should now be completed.

Implementing virtual memory

You will now implement your own virtual memory management system. We have seen in class the many different logical components that are allocated in memory for each process. In the context of this TP, we will SOLELY focus on the part of the memory that holds the dynamic memory used in functions, i, the heap. In particular, the goal is to expose three functions to the application:

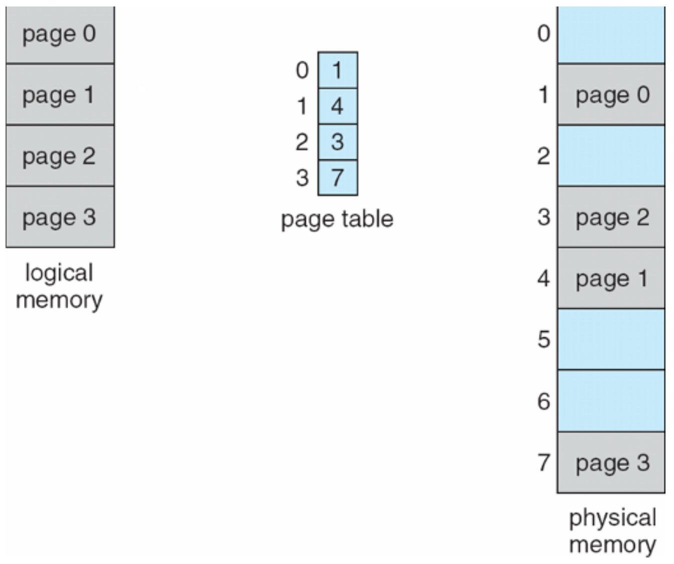
- `void *my_malloc(int size)`: similarly the real world `malloc`, your `malloc` allocates a given amount of memory for the application.
- `void my_free(void *data)`: frees the memory.
- `void my_copy(void *mem, void *data, int size)`: allows writing raw bytes to memory. Ideally, your `copy` function takes as input a memory pointer to your virtual memory, a pointer to the data you want to copy, and the size of memory to copy.

NOTE: This implementation does not (obviously) replicate the exact implementation of the Linux memory management system. In particular, because of the simpler version of our solution, the interaction with the swap occurs only when copies to memory occur, rather than due to context switches. But alas, we're here to learn 😀

How to implement the virtual memory?

Remember from the course that the virtual memory management is based on two core abstractions: (1) the logical map between application allocated pages and pages in the physical memory; (2) a linked list to refer to the blocks of memory obtained through `malloc`.

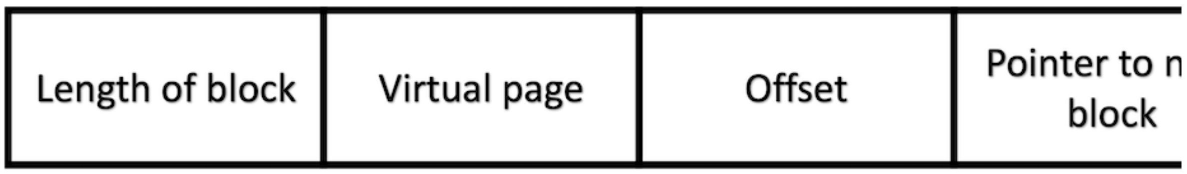
Logical map. The logical map enables the translation between in use pages and their location in memory as shown in the figure below:



This enables to map virtual addresses to the their physical location. In the context of your exercise, this means that every address returned by `my_malloc` are not the actual physical memory address but rather a logical abstraction. For example considering the figure, if `my_malloc` returns the value 0 inside page 0, this equals to a 0 offset with respect to the addresses in page 0. When looking at the page table, you will find out that page 0 is physically located in page 1 in the physical memory still with a 0 offset. In short, virtual address 0 corresponds to the physical address $\text{start\_of\_page\_1}+0$.

Implement the function `virtual_to_physical` to translate virtual addresses to physical ones. To do so, you will (most likely) need to maintain a data structure that enables to connect virtual page numbers to physical ones. A vector should do the trick.

Linked list. Memory blocks allocated using malloc are normally smaller than a page. For the sake of exercise, let's assume that it is not possible to request a memory block bigger than a page, i, they have to be smaller than 4096B. The content of your linked list will connect elements with information resembling the following figure:



Note that in class we saw that OSs normally use double linked list but for the sake of this exercise a simple linked list will suffice. Any time a function that manipulates a function is invoked (`my_malloc`, `my_free`, or `my_copy`), the listed is traversed to find the appropriate area of memory that is being manipulated.

The last step

You now should have all the notions to implement the three requested functions: `my_malloc`, `my_free`, and `my_copy`. Recollect that no direct memory access is permitted in this exercise and all write operations are performed via the `my_copy` function.

NOTE: the sole constrain we have on the size of allocated blocks is that they have to be smaller than the max page size. This means that allocations will use smaller portions of a page. Re-utilization of freed blocks is permitted. But no need to refactor the location of memory to avoid waste of unused portions of memory.

What about the swap?

Remember from class that your virtual memory is usually bigger than the actual physical memory. This means that, if the physical memory runs out, you will start having to swap pages back and forth from swap. The extra bonus for this TP is for who accounts for this problem and readily handles memory in the presence of swap.

NOTE: No strict requirement is imposed on the replacement policy.

SUBMISSION INSTRUCTIONS

Submit all the implemented code (including the TD 5 solution) using the dedicated submission page by Sunday 31/3 at 19h. One point will be detract for every 24h of delay after the deadline.

Modifié le: vendredi 22 mars 2024, 08:24

◄ TD 6 - Processus

ADMINISTRATION

› Administration du cours