# Assignment9

11911839 聂雨荷

## Q1

[50pts] Read Chapter 21 of "Three Easy Pieces" (https://pages.cs.wisc.edu/~remzi/OSTEP/vm -beyondphys.pdf ) and explain what happens when the process accesses a memory page not present in the physical memory.

- When the hardware looks in the PTE, it may find that the page is not present in physical memory. The hardware determines this is though the present bit. If it is set to **zero**, the page is not in memory but rather on disk somewhere. The act of accessing a page that is not in physical memory is commonly referred to as a **page fault**.
- Upon a page fault, the OS is invoked to service the page fault. A particular piece of code, known as a **page-fault handler**, runs, and must service the page fault.
- If a page is not present and has been swapped to disk, the OS will need to **swap the page into memory in order to service the page fault**. When the OS receives a page fault for a page, it looks in the **PTE** to find the **address**, and issues the **request to disk** to fetch the page into memory.
- This next attempt may generate a **TLB miss**, which would then be serviced and **update the TLB with the translation** (one could alternately update the TLB when servicing the page fault to avoid this step).
- Finally, a last restart would **find the translation in the TLB** and thus **proceed to fetch the desired data or instruction from memory** at the translated **physical address**.
- Note that while the I/O is in flight, the process will be in the blocked state. Thus, the OS will be free to run other ready processes while the page fault is being serviced. Because I/O is expensive, this overlap of the I/O (page fault) of one process and the execution of another is yet another way a multiprogrammed system can make the most effective use of its hardware.
- If the memory is **full**, the OS might like to first **page out** one or more pages to make room for the new page(s) the OS is about to bring in. The process of picking a page to kick out, or **replace** is known as the **page-replacement policy**.

Now three important cases to understand when a TLB miss occurs.

- First, that the page was both present and valid ; in this case, the TLB miss handler can simply grab the PFN from the PTE, retry the instruction (this time resulting in a TLB hit), and thus continue as described (many times) before.
- In the second case, the page fault handler must be run; although this was a legitimate page for the process to access (it is valid, after all), it is not present in physical memory.
- Third (and finally), the access could be to an invalid page, due for example to a bug in the program. In this case, no other bits in the PTE really matter; the hardware traps this invalid access, and the OS trap handler runs, likely terminating the offending process.

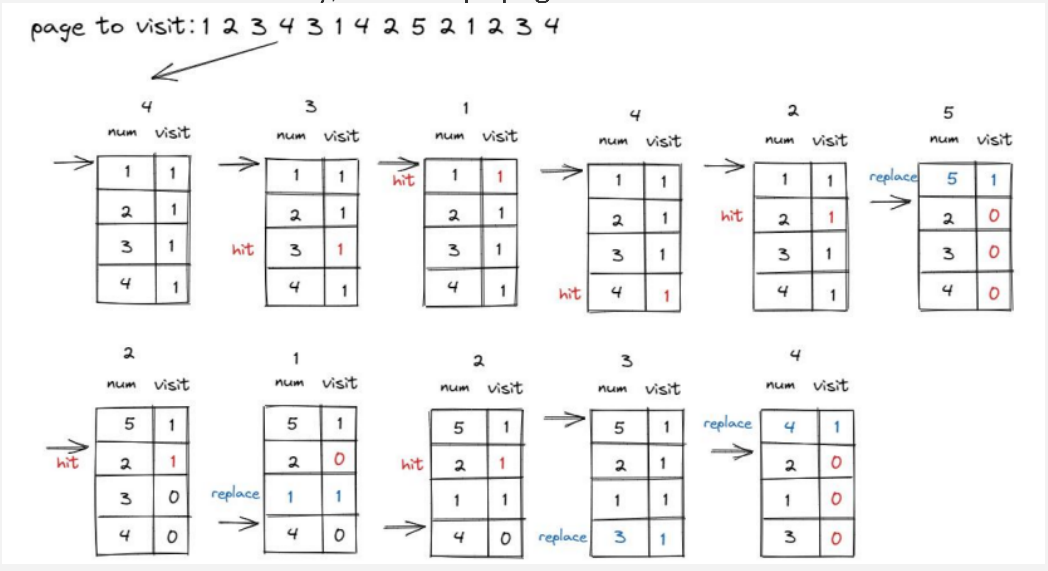
What the OS roughly must do in order to service the page fault.

- First, the OS must find a physical frame for the soon-to-be-faulted-in page to reside within;
- if there is no such page, we'll have to wait for the replacement algorithm to run and kick some pages out of memory, thus freeing them for use here.

## Q2

[50pts] Realize Clock algorithm in `swap_clock.c`

Clock: Clock algorithm organizes pages into a circular linked list, just like a clock. The pointer points to the pages loaded the earliest. Besides, Clock algorithm requires a flag bit in PTE to indicate if the corresponding page is used. When the page is used, MMU in CPU will set the flag to 1. When system needs to replace a page, system read the PTE pointer points to, and replace it if the flag is 0, otherwise read the next PTE. The algorithm demonstrates the idea of LRU, and is easy to implement and costs less, but require hardware support to set an reference bit. Clock algorithm is the same as LRU essentially, but it skips pages with reference bit of 1.



Code you may need:

```
list_entry_t *curr_ptr;
*ptr_page = le2page(le, pra_page_link);
pte_t* ptep = get_pte(mm->pgdir, ptr_page->pra_vaddr, 0);
bool accessed = *pte & PTE_A;
```

Please realize algorithm in `swap_clock.c`. Change `sm` in `swap.c` to clock to test your code. Please take screen-shot of your code(with annotations) and the running result.

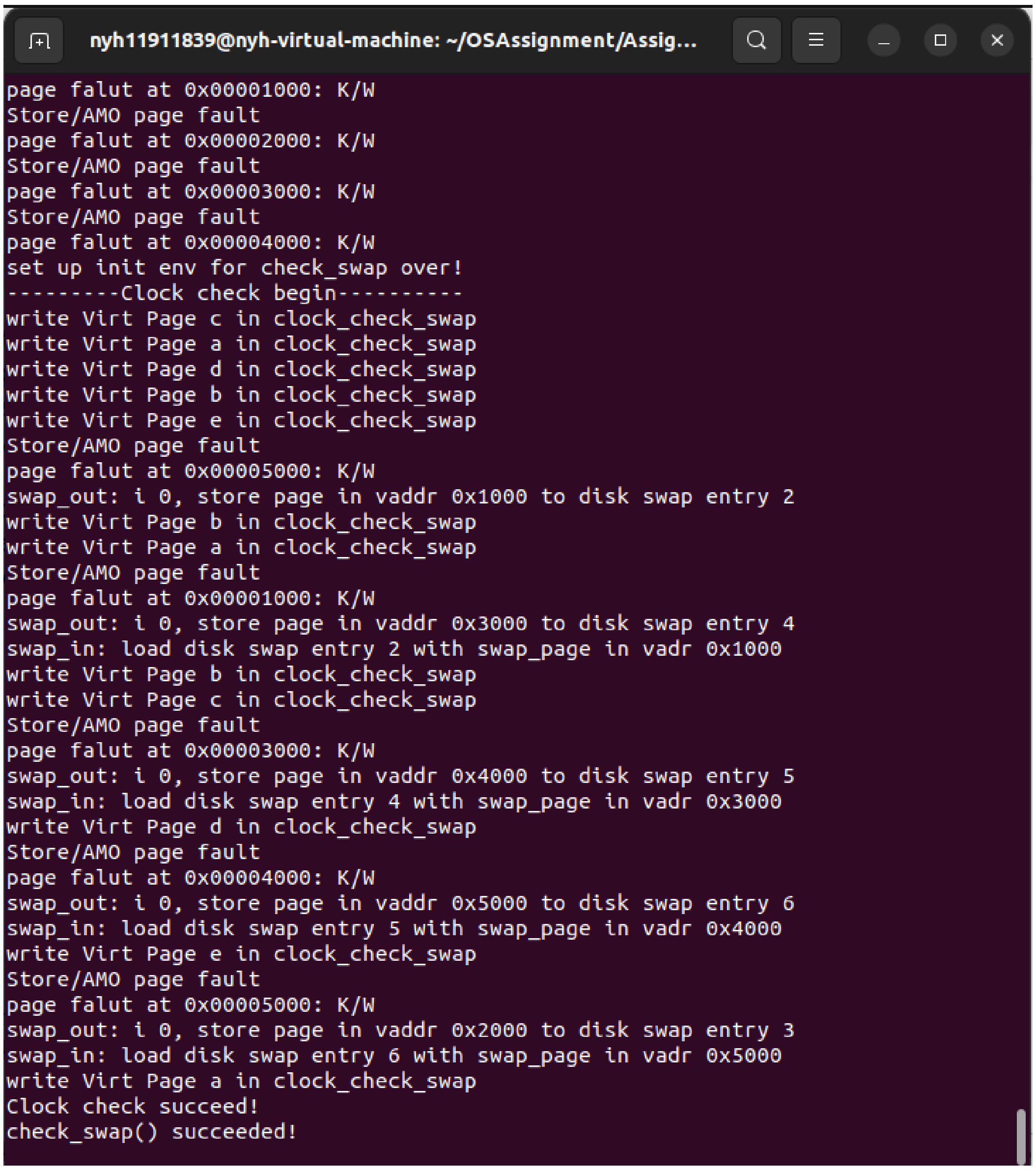
code in `swap.c`

```
static int
_clock_init_mm(struct mm_struct *mm)
{
    // initialize the pra_list
    list_init(&pra_list_head);
    mm->sm_priv = &pra_list_head;

    // initially, set the curr_ptr to the first page
    curr_ptr = &pra_list_head;
    return 0;
}
```

```
static int
_clock_map_swappable(struct mm_struct *mm, uintptr_t addr,
struct Page *page, int swap_in)
{
    // insert the new page to the prev entry of the
curr_ptr
    list_entry_t *entry=&(page->pra_page_link);
    assert(entry != NULL && curr_ptr != NULL);
    list_add_before(curr_ptr, entry);
    return 0;
}
```

```
static int
_clock_swap_out_victim(struct mm_struct *mm, struct Page **
ptr_page, int in_tick)
{
    list_entry_t *head=(list_entry_t*) mm->sm_priv;
    assert(head!=NULL && in_tick==0);

    // start from the next element
    // curr_ptr = curr_ptr->next;
    while(1){
        // if curr_ptr = head, jump to the next one
        if(curr_ptr != head){
            *ptr_page = le2page(curr_ptr, pra_page_link);
            pte_t* ptep = get_pte(mm->pgdir, (*ptr_page)-
>pra_vaddr, 0);
            if((*ptep) & PTE_A){ // visit bit = 1
                (*ptep) &= (~PTE_A); // set to 0
            }
            else{ // visit bit = 0 -> delete the victim
                curr_ptr = curr_ptr->next;
                list_del(curr_ptr->prev);
                break;
            }
        }
        curr_ptr = curr_ptr->next;
    }
    return 0;
}
```

running result



```
page falut at 0x00001000: K/W
Store/AMO page fault
page falut at 0x00002000: K/W
Store/AMO page fault
page falut at 0x00003000: K/W
Store/AMO page fault
page falut at 0x00004000: K/W
set up init env for check_swap over!
---------Clock check begin----------
write Virt Page c in clock_check_swap
write Virt Page a in clock_check_swap
write Virt Page d in clock_check_swap
write Virt Page b in clock_check_swap
write Virt Page e in clock_check_swap
Store/AMO page fault
page falut at 0x00005000: K/W
swap_out: i 0, store page in vaddr 0x1000 to disk swap entry 2
write Virt Page b in clock_check_swap
write Virt Page a in clock_check_swap
Store/AMO page fault
page falut at 0x00001000: K/W
swap_out: i 0, store page in vaddr 0x3000 to disk swap entry 4
swap_in: load disk swap entry 2 with swap_page in vadr 0x1000
write Virt Page b in clock_check_swap
write Virt Page c in clock_check_swap
Store/AMO page fault
page falut at 0x00003000: K/W
swap_out: i 0, store page in vaddr 0x4000 to disk swap entry 5
swap_in: load disk swap entry 4 with swap_page in vadr 0x3000
write Virt Page d in clock_check_swap
Store/AMO page fault
page falut at 0x00004000: K/W
swap_out: i 0, store page in vaddr 0x5000 to disk swap entry 6
swap_in: load disk swap entry 5 with swap_page in vadr 0x4000
write Virt Page e in clock_check_swap
Store/AMO page fault
page falut at 0x00005000: K/W
swap_out: i 0, store page in vaddr 0x2000 to disk swap entry 3
swap_in: load disk swap entry 6 with swap_page in vadr 0x5000
write Virt Page a in clock_check_swap
Clock check succeed!
check_swap() succeeded!
```