CS334 Assignment 4 Report

SID: 12012919

Name: 廖铭骞

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.

Firstly, the OS will find a physical frame for the soon-to-be-faulted-in page to reside within. If it dosen't have the physical frame, the replacement algorithm will run and kick some pages out of memory, thus freeing them for use here.

After we have a physical frame in hand, then the handler will issue the I/O request to read in the page from the swap space in disk.

Finally, when that slow operation completes, the OS updates the page table and retries the instruction. The retry will result in a TLB miss, and then, upon another try, the TLB will be hit, the the hardware will be able to access the value it want to access.

Consider the following reference string `7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1`. Consider 4 pages. What are the number of page faults with the following policy: Optimal (MIN), LRU, FIFO.

For the Optimal (MIN) policy: 8 page faults

For the LRU policy: 8 page faults

For the FIFO policy: 10 page faults

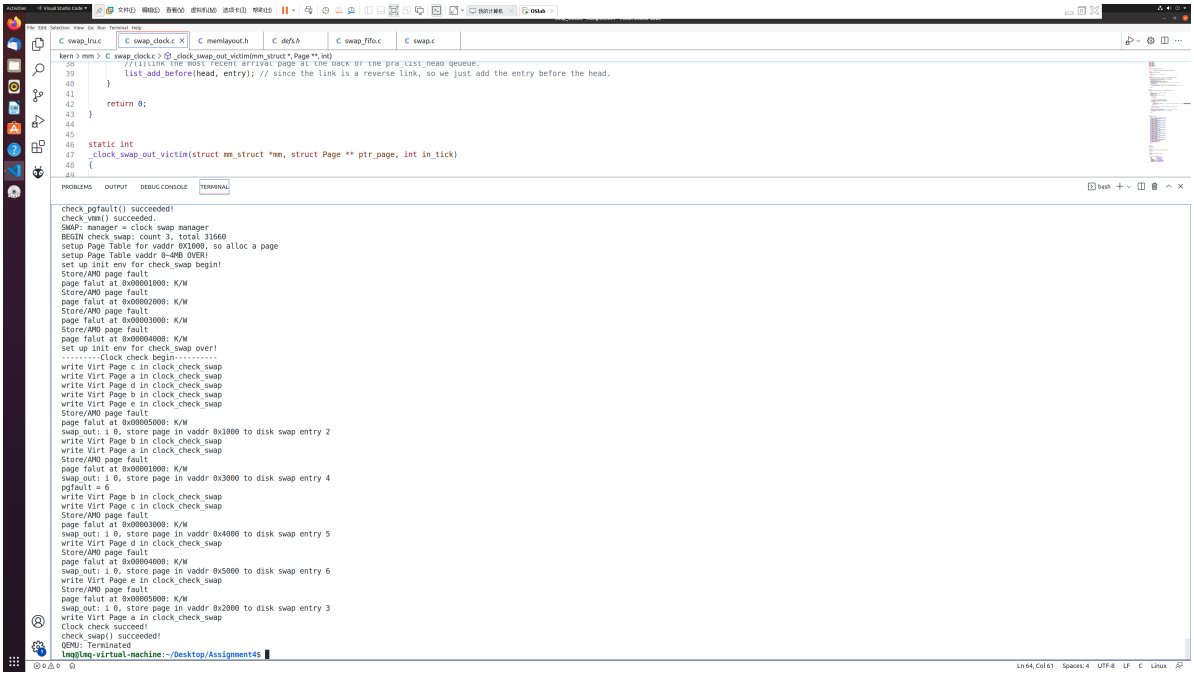
Realiza Clock algorithm in `swap_clock.c`

Screen-shot

```
11  list_entry_t pra_list_head, *curr_ptr = NULL;
12
13  /**
14  * init pra_list_head and let  mm->sm_priv point to the addr of pra_list_head.
15   *             Now, From the memory control struct mm_struct, we can access clock PRA
16   */
17  static int
18  _clock_init_mm(struct mm_struct *mm)
19  {
20      //TODO
21      mm -> sm_priv = NULL; // set the head node to NULL
22      return 0;
23  }
24
25  static int
26  _clock_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page *page, int swap_in)
27  {
28      list_entry_t *head = (list_entry_t *)mm->sm_priv; // get the head node.
29      list_entry_t *entry = &(page->pra_page_link); // get the current entry.
30      assert(entry != NULL);
31      // record the page access situation
32      pte_t *ptep = get_pte(mm->pgdir, page->pra_vaddr, 0);
33      *ptep = *ptep & PTE_A; // set the acccess bit to accessed
34      if(head == NULL){ // currently, there are no head, so that we should let the entry be the head
35          list_init(entry);
36          mm->sm_priv = entry;
37      }else{
38          //(1)link the most recent arrival page at the back of the pra_list_head qeueue.
39          list_add_before(head, entry); // since the link is a reverse link, so we just add the entry before the head.
40      }
41
42      return 0;
43  }
44
```

```
45
46  static int
47  _clock_swap_out_victim(struct mm_struct *mm, struct Page ** ptr_page, int in_tick)
48  {
49
50      //TODO
51      list_entry_t *head = (list_entry_t*)(mm -> sm_priv);
52      assert(head != NULL);
53      list_entry_t *victim = 0;
54      *ptr_page = 0;
55      while(head != victim){ // iterate the link list.
56
57          if(victim == 0)
58          {
59              victim = head;
60          }
61
62          struct Page *page = le2page(victim, pra_page_link);
63          pte_t *ptep = get_pte(mm->pgdir, page->pra_vaddr, 0);
64          bool accessed = *ptep & PTE_A; // get the access bit
65          if(!accessed)
66          {
67              *ptr_page = page;
68              mm->sm_priv = list_next(victim); // first set the position of head pointer and delete the victim page.
69              list_del(victim);
70              break;
71          }
72
73          *ptep = *ptep & ~PTE_A; // if the access bit is accessed, then it should be cleared
74          victim = list_next(victim);
75      }
76
77      if(*ptr_page == 0){ // if cannot find the victim, then we should replace the head page
78          *ptr_page = le2page(head, pra_page_link);
79          mm->sm_priv = list_next(head);// set the new position of head page.
80          list_del(head);
81      }
82      return 0;
83
84
85  }
86
87
```

Running Result:



Realize LRU algorithm in `swap_lru.c`

Screen Shot:

```
list_entry_t pra_list_head, *curr_ptr2 = NULL;

static int
_lru_init_mm(struct mm_struct *mm)
{
    // TODO
    list_init(&pra_list_head);
    mm->sm_priv = &pra_list_head;
    // cprintf(" mm->sm_priv %x in fifo_init_mm\n",mm->sm_priv);
    return 0;
    // mm->sm_priv = NULL; // set the head node to NULL
    // return 0;
}
```

```c
static int
_lru_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page *page, int swap_in)
{
    // // TODO
    list_entry_t *head = (li                                priv;
    list_entry_t *entry = &(                                ;

    assert(entry != NULL && head != NULL);
    // record the page access situlation

    //(1)link the most recent arrival page at the back of the pra_list_head qeueue.
    // (*(unsigned char *)page->pra_vaddr) ++;
    list_add(head, entry);
    return 0;
```

```
list_entry_t *head
// TODO
```

```c
static int
_lru_swap_out_victim(struct mm_struct *mm, struct Page **ptr_page, int in_tick)
{
    // TODO
    list_entry_t *head = (list_entry_t *)mm->sm_priv;
    assert(head != NULL);
    assert(in_tick == 0);
    list_entry_t *curr_ptr;
    curr_ptr = list_next(head);
    list_entry_t *e = list_next(head);
    unsigned int min = -1;
    cprintf("11111111111111111111111111111\n");
    while (curr_ptr != head)
    {
        cprintf("222222\n");
        struct Page *p = le2page(curr_ptr, pra_page_link);
        cprintf("%-llx\n", p);
        unsigned int c = *(unsigned int *)p->pra_vaddr;
        if (c < min)
        {
            min = c;
            *ptr_page = p;
            e = curr_ptr;
        }
        unsigned int c = *(unsigned int *)p->pra_vaddr;
        if (c < min)
        {
            min = c;
            *ptr_page = p;
            e = curr_ptr;
        }
        curr_ptr = list_next(curr_ptr);
    }
    cprintf("%d\n", e);
    list_del(e);
    return 0;
    // TODO
    // list_entry_t *head = (list_entry_t *)(mm->sm_priv);
    // assert(head != NULL);
    // list_entry_t *current_ptr = 0;
    // list_entry_t *victim = 0;
    // *ptr_page = 0;
    // unsigned int min = 100;
```

Running Result:

```
-1071494928
swap_out: i 0, store page in vaddr 0x3000 to disk swap entry 4
swap_in: load disk swap entry 3 with swap_page in vadr 0x2000
write Virt Page 3 in lru_check_swap
Store/AMO page fault
page falut at 0x00003000: K/W
11111111111111111111111111111
222222
ffffffffc02248c0
222222
ffffffffc0224908
222222
ffffffffc0224998
222222
ffffffffc0224950
-1071494712
swap_out: i 0, store page in vaddr 0x1000 to disk swap entry 2
swap_in: load disk swap entry 4 with swap_page in vadr 0x3000
LRU check succeed!
check_swap() succeeded!
```