

Assignment 9

12011702 张镇涛

1

The act of accessing a page that is not in physical memory is commonly referred to as a page fault.

According to the following paragraphs:

Recall that with TLB misses, we have two types of systems: hardware-managed TLBs (where the hardware looks in the page table to find the desired translation) and software-managed TLBs (where the OS does). In either type of system, if a page is not present, the OS is put in charge to handle the page fault. The appropriately-named OS **page-fault handler** runs to determine what to do. Virtually all systems handle page faults in software; even with a hardware-managed TLB, the hardware trusts the OS to manage this important duty.

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. Thus, a question arises: how will the OS know where to find the desired page? In many systems, the page table is a natural place to store such information. Thus, the OS could use the bits in the PTE normally used for data such as the PFN of the page for a disk address. 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.

When the disk I/O completes, the OS will then update the page table to mark the page as present, update the PFN field of the page-table entry (PTE) to record the in-memory location of the newly-fetched page, and retry the instruction. 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.

In the process described above, you may notice that we assumed there is plenty of free memory in which to **page in** a page from swap space. Of course, this may not be the case; memory may be full (or close to it). Thus, 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**.

The following procedure is performed:

1. A particular piece of code, known as a page-fault handler runs.
2. The OS looks in the PTE to find the address, and issues the request to disk to swap the page into memory.
3. One or more pages will be paged out to make room for the new page(s) the OS is about to bring in if memory is full, and the page which needed to swap out is based on page-replacement policy.
4. The process is blocked and disk perform I/O task.
5. The page table is updated and the process is restarted so that it could get the desired data.

Code (Screenshot)

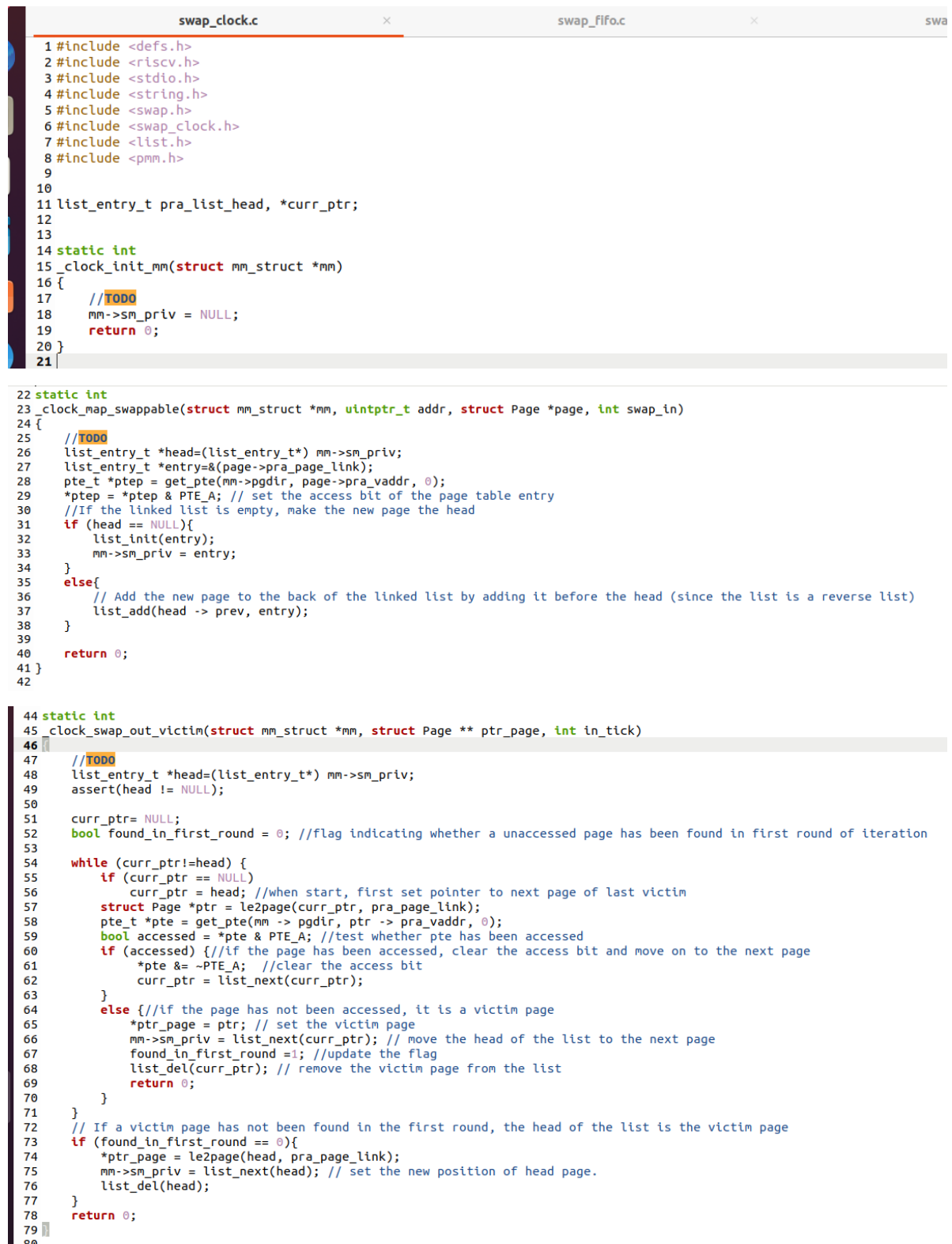
First, we need to modify `swap.c` to redirect to `swap_manager_clock`.

```

5 //-----
7 //sm = &swap_manager_fifo;//use first in first out Page Replacement Algorithm
3
9 //exercise1
9 sm = &swap_manager_clock;
1

```

The `swap_clock.c` is implemented as follows:



```

swap_clock.c
swap_fifo.c
swa

1 #include <defs.h>
2 #include <riscv.h>
3 #include <stdio.h>
4 #include <string.h>
5 #include <swap.h>
6 #include <swap_clock.h>
7 #include <list.h>
8 #include <pmm.h>
9
10
11 list_entry_t pra_list_head, *curr_ptr;
12
13
14 static int
15 _clock_init_mm(struct mm_struct *mm)
16 {
17     //TODO
18     mm->sm_priv = NULL;
19     return 0;
20 }
21
22 static int
23 _clock_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page *page, int swap_in)
24 {
25     //TODO
26     list_entry_t *head=(list_entry_t*) mm->sm_priv;
27     list_entry_t *entry=&(page->pra_page_link);
28     pte_t *ptep = get_pte(mm->pgdir, page->pra_vaddr, 0);
29     *ptep = *ptep & PTE_A; // set the access bit of the page table entry
30     //If the linked list is empty, make the new page the head
31     if (head == NULL){
32         list_init(entry);
33         mm->sm_priv = entry;
34     }
35     else{
36         // Add the new page to the back of the linked list by adding it before the head (since the list is a reverse list)
37         list_add(head -> prev, entry);
38     }
39
40     return 0;
41 }
42
43
44 static int
45 _clock_swap_out_victim(struct mm_struct *mm, struct Page ** ptr_page, int in_tick)
46 {
47     //TODO
48     list_entry_t *head=(list_entry_t*) mm->sm_priv;
49     assert(head != NULL);
50
51     curr_ptr= NULL;
52     bool found_in_first_round = 0; //flag indicating whether a unaccessed page has been found in first round of iteration
53
54     while (curr_ptr!=head) {
55         if (curr_ptr == NULL)
56             curr_ptr = head; //when start, first set pointer to next page of last victim
57         struct Page *ptr = le2page(curr_ptr, pra_page_link);
58         pte_t *pte = get_pte(mm -> pgdir, ptr -> pra_vaddr, 0);
59         bool accessed = *pte & PTE_A; //test whether pte has been accessed
60         if (accessed) { //if the page has been accessed, clear the access bit and move on to the next page
61             *pte &= ~PTE_A; //clear the access bit
62             curr_ptr = list_next(curr_ptr);
63         }
64         else { //if the page has not been accessed, it is a victim page
65             *ptr_page = ptr; // set the victim page
66             mm->sm_priv = list_next(curr_ptr); // move the head of the list to the next page
67             found_in_first_round = 1; //update the flag
68             list_del(curr_ptr); // remove the victim page from the list
69             return 0;
70         }
71     }
72     // If a victim page has not been found in the first round, the head of the list is the victim page
73     if (found_in_first_round == 0){
74         *ptr_page = le2page(head, pra_page_link);
75         mm->sm_priv = list_next(head); // set the new position of head page.
76         list_del(head);
77     }
78     return 0;
79 }
80

```

Reference: https://blog.csdn.net/weixin_43995093/article/details/101688540

Running Result

```
os12011702@vmos-tony: ~/oslab/Assignment9

setup Page Table for vaddr 0x1000, so alloc a page
setup Page Table vaddr 0~4MB OVER!
set up init env for check_swap begin!
Store/AMO page fault
page fault at 0x00001000: K/W
Store/AMO page fault
page fault at 0x00002000: K/W
Store/AMO page fault
page fault at 0x00003000: K/W
Store/AMO page fault
page fault 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 fault 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 fault at 0x00001000: K/W
swap_out: i 0, store page in vaddr 0x3000 to disk swap entry 4
write Virt Page b in clock_check_swap
write Virt Page c in clock_check_swap
Store/AMO page fault
page fault at 0x00003000: K/W
swap_out: i 0, store page in vaddr 0x4000 to disk swap entry 5
write Virt Page d in clock_check_swap
Store/AMO page fault
page fault at 0x00004000: K/W
swap_out: i 0, store page in vaddr 0x5000 to disk swap entry 6
write Virt Page e in clock_check_swap
Store/AMO page fault
page fault at 0x00005000: K/W
swap_out: i 0, store page in vaddr 0x2000 to disk swap entry 3
write Virt Page a in clock_check_swap
Clock check succeed!
check_swap() succeeded!
```