ICS EXE 14

June 10, 2020

1 Address Translation

This problem concerns the way virtual addresses are translated into physical addresses. Below are the specifications of the system on which the translation occurs:

- The main memory is byte addressable.
- The memory accesses are to **1-byte** words (not 4-byte words).
- The system uses **one-level** page table.
- Virtual addresses are **25 bits** wide.
- PPN is **9 bits** wide.
- The amount of PTE is 2^{14}.
- The TLB is **4-way** set associative with **16** total entries.

1.1

The VPO bits	11
The physical address bits	20
The page size	2KB
The number of bits for TLBT	12

The contents of the TLB and first 16 entries of the page table are given below. All numbers are in hexadecimal. According to the illustration and answer the questions. Please fill in the blanks.

Set	Tag	PPN	Valid	Tag	PPN	Valid
0	0CD	09	1	AA1	00	1
	3E0	62	0	C4C	48	1
1	312	45	0	010	75	1
	987	3A	1	D39	3F	0
2	038	E3	0	0A7	13	1
	18B	52	1	49B	11	0
3	6C0	42	0	075	50	0
	013	39	1	0F2	0D	0

TLB: 4 sets, 16 entries, 4-way set associative

VPN	PPN	Valid	VPN	PPN	Valid
00	39	1	08	OD	0
01	52	1	09	45	0
02	E3	0	0A	13	1
03	00	1	0B	3A	1
04	11	0	0C	09	1
05	50	0	0D	42	0
06	62	0	0E	3F	0
07	75	1	0F	48	1

Page Table: Only the first 16 PTEs are shown

1.2

Please translate virtual address to physical address and get the data from cache if hit (otherwise entering '–')

Parameter	Value
Virtual Address	0x014F2D3
VPN	0x29E
TLB Index	0x2
TLB Tag	0xA7
TLB Hit? (Y/N)	Y
Page Fault? (Y/N)	N
PPN	0x13
Physical Address	0x09AD3

Parameter	Value
Virtual Address	0x07C01A5
VPN	0x0F80
TLB Index	0x0
TLB Tag	0x3E0
TLB Hit? (Y/N)	N
Page Fault? (Y/N)	-
PPN	0x-
Physical Address	0x-

2 Virtual Address

Suppose the program runs on the Intel Core i7/Linux Memory System discussed in section 9.7 of the CSAPP. Please answer the following questions.

```
#include <unistd.h>
#include <sys/mman.h>
#include <stdio.h>
#define ARRAY_LEN (125*1024)
int main(void) {
  int idx;
  int fd = open("ics.txt",O_RDWR);
A:
  char* sbuf = mmap(0, ARRAY_LEN, PROT_READ | PROT_WRITE,
                    MAP_SHARED, fd, 0);
  char* pbuf = mmap(0, ARRAY_LEN, PROT_READ | PROT_WRITE,
                    MAP_PRIVATE, fd, 0);
B:
  for (idx = 0; idx < ARRAY_LEN; idx++) {
    sbuf[idx] = sbuf[idx] + 1;
    pbuf[idx] = pbuf[idx] + 1;
  }
C:
  fork();
D:
  for (idx = 0; idx < ARRAY_LEN; idx++) {
    pbuf[idx] = pbuf[idx] + 1;
  }
E:
  munmap(sbuf, ARRAY_LEN);
  munmap(pbuf, ARRAY_LEN);
  close(fd);
  return 0;
}
```

Assumption:

1) After setting **MAP_SHARED** flag, the updates to the shared mapping area will be immediately synchronized to the file.

2) After setting **MAP_PRIVATE** flag, the changes made to the file after **mmap()** call and before copy-on-write are visible to the mapped area. The modification on the mapped area memory will cause a copy-on-write (COW) mapping.

3) The **ics.txt** is a file filled with a lot of character '0' and its size is more

than **ARRAY_LEN**.

The address of **sbuf** is **0xb7be2000** and that of **pbuf** is **0xb7cd4000** before label **B**. Please answer the following questions (**NOTE** please ignore the page fault on the **stack** for the following questions)

2.1

How many page fault exception are raised by code between label B and C? How many of them will handle COW? Please also write down your explanation.

96, 32

Reason: Because mmap will not copy data from disk to memory. To fill the page table at the first access, any read to sbuf and pbuf will cause page fault after mmap. And the write to pbuf will cause COW for it's a private mapping. And the content of array sbuf and pbuf are in 32 pages.

2.2

How many page fault exception are raised by code between label D and E of the parent process? How many of them will handle COW? Please also write down your explanation.

32, 32

After fork, all the pages are set as readonly and all the write to memory sbuf will cause COW. And the content of array pbuf are in 32 pages.

2.3

4. Please write down the value of below variables of process when the program reach label C.

sbuf[0] = '1' **sbuf[1]** = '1'
pbuf[0] = '2' **pbuf[1]** = '1'

3 Replacement Policy

Assume we have a primary device with 3 physical blocks, please complete the following questions.

3.1

Suppose we are using LRU replacement policy, please complete the following table. (**NOTE**: you do not need to consider the order of primary device contents).

Reference String	1	2	3	4	2	1	3	1	4
Primary Device Contents	1	1	1	2	2	1	1	1	1
	-	2	2	3	3	2	2	2	3
	-	-	3	4	4	4	3	3	4

3.2

Suppose we are using clock algorithm with rules:

- The clock pointer points to position 0 initially.
- We do not reset the clock pointer after we have found an evict page, so next time we start from the position after previous victim.
- Each page brought in were set as accessed initially.

Please complete the following table. (**NOTE**: use "*" to represent current clock pointer).

Reference String	1	2	3	4	2	1	3	1	4
Primary Device Contents	1	1	1*	4	4	4*	4	4	4
	*	2	2	2*	2*	2	3	3	3
	-	*	3	3	3	1	1*	1*	1*

3.3

Why do we use clock algorithm instead of directly implement LRU policy in most realistic systems? Please give your reason.

a) On many workloads, the performance of clock algorithm is close to that of LRU.

b) Clock algorithm is easier to implement. To implement the LRU policy, we need extra accounting work on every memory reference, and extra space to store information related to the access order (like an access timestamp or an extra access stack) and extra time to get the last recent page (like sorting of timestamps and updating the stack).

So clock algorithm is wildly used as an approximating of LRU policy. It takes history into consideration while keeping an acceptable time and space overhead.