HCMC University Of Technology
Faculty of Computer Science & Engineering

Lab 7 Paging
Course: Operating Systems

May 8, 2021

Goal: The objective of this lab is simulating the process of paging in memory.

1 Background

Segmentation (a generalization of dynamic relocation) helped us do this, but has some problems; in particular,managing free space becomes quite a pain as memory becomes fragmented and segmentation is not as flexible as we might like. Is there a better solution?

Thus comes along the idea of paging, Instead of splitting up our address space into three logical segments (each of variable size), we split up our address space into fixed-sized units we call a **page**.

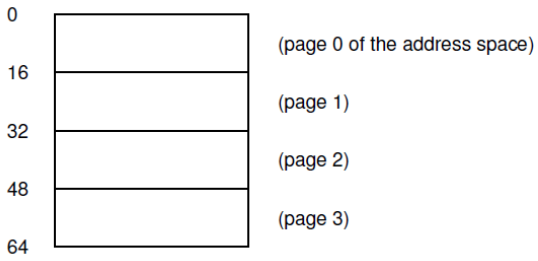

Figure 1.1: A Simple 64-byte Address Space

Figure 1.1 is an example of a tiny address space, only 64 bytes total in size, with 16 byte pages (real address spaces are much bigger, of course, commonly 32 bits and thus 4-GB of address space, or even 64 bits).

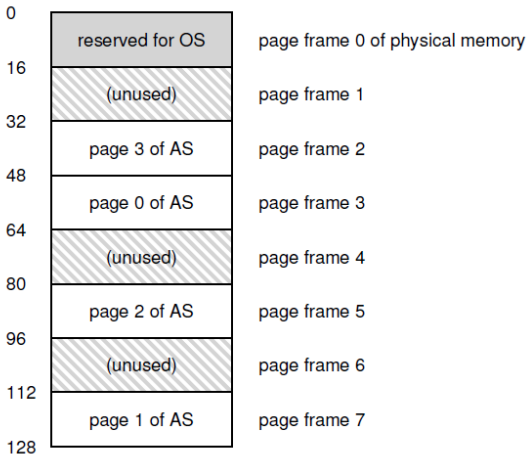

Figure 1.2: 64-Byte Address Space Placed In Physical Memory

Thus, we have an address space that is split into four pages (0 through 3). With paging, physical memory is also split into some number of pages as well; we sometimes will call

each page of physical memory a page frame like Figure 1.2.
Paging, as we will see, has a number of advantages over our previous approaches:

- flexibility (**why?**)
- simplicity (**why?**)

To record where each virtual page of the address space is placed in physical memory, the operating system keeps a per-process data structure known as a page table. The major role of the page table is to store address translations for each of the virtual pages of the address space, thus letting us know where in physical memory they live.

We know enough to perform an address-translation example. Let's imagine the process with that tiny address space (64 bytes) is performing a memory access.

To translate this virtual address that the process generated, we have to first split it into two components: the virtual page number (VPN), and the offset within the page. For this example, because the virtual address space of the process is 64 bytes,we need 6 bits total for our virtual address ($26 = 64$). Thus, our virtual address:

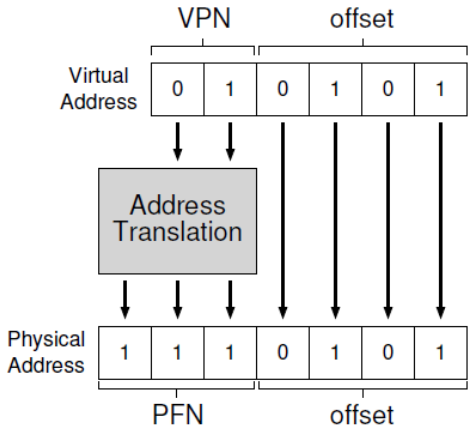

Figure 1.3: The Address Translation Process

Paging: Faster Translations (TLBs) How to speed up address translation

- How can we speed up address translation, and generally avoid the extra memory reference that paging seems to require?
- What hardware support is required?
- What OS involvement is needed?

To speed address translation, we are going to add what is called (for historical reasons [CP78]) a **translation-lookaside buffer**, or TLB. A TLB is part of the chip's memory-management unit (MMU), and is simply a hardware cache of popular virtual-to-physical address translations; thus, a better name would be an address-translation cache.

TLB Basic Algorithm

```
1  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2  (Success, TlbEntry) = TLB_Lookup(VPN)
3  if (Success == True) // TLB Hit
4          if (CanAccess(TlbEntry.ProtectBits) == True)
5                  Offset = VirtualAddress & OFFSET_MASK
6                  PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7                  AccessMemory(PhysAddr)
8          else
9                  RaiseException(PROTECTION_FAULT)
10 else // TLB Miss
11         PTEAddr = PTBR + (VPN * sizeof(PTE))
12         PTE = AccessMemory(PTEAddr)
13         if (PTE.Valid == False)
14                 RaiseException(SEGMENTATION_FAULT)
15         else if (CanAccess(PTE.ProtectBits) == False)
16                 RaiseException(PROTECTION_FAULT)
17         else
18                 TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
19                 RetryInstruction()
```

2 Practice

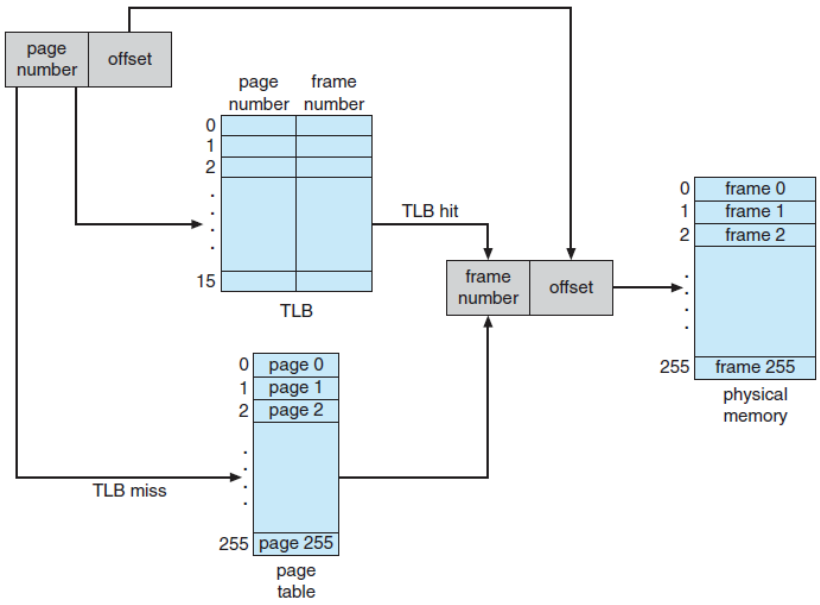

Figure 2.1: A representation of the address-translation process

Student need to complete the given source code with "TO DO" part. This program aims to simulate the process of translating virtual memory into physical memory with TLB algorithm. The layout of this program below:

```
1  #include <stdio.h>
2  #include <stdint.h>
3
4  #define TLB_SIZE 4
5  #define PAGE_TABLE_SIZE 10
6  #define OFF_LEN 12
7
8  typedef uint32_t address_t;
9  typedef uint32_t index_t;
10 typedef uint32_t offset_t;
11
12 struct tlb_slot_t {
13         index_t virtual; // Index for virtual address
14         index_t physical; // Index for physical address
15         index_t count; // Number of times this slot is accessed
16 };
17
18 struct tlb_slot_t tlb[TLB_SIZE];
```

```
19
20 struct tlb_slot_t page_table[PAGE_TABLE_SIZE];
21
22 /* Get index part of an address */
23 index_t get_index(address_t addr);
24
25 /* Get offset part of an address */
26 offset_t get_offset(address_t addr);
27
28 /* Get the least accessed slot from TLB */
29 struct tlb_slot_t * get_least_accessed_slot();
30
31 /* Translate virtual address to physical address */
32 int translate(address_t virtual, address_t * physical) {
33         // TODO: Translate virtual address to physical address.
34         // Return 1 if the virtual address is valid, otherwise, return 0
35         // If the virtual address is valid, writing its physical counterpart
36         // to physical address.
37         return 0;
38
39 }
40
41 int main() {
42         /* Init page table */
43         page_table[0] = (struct tlb_slot_t) {0x00001, 0x52354, 0};
44         page_table[1] = (struct tlb_slot_t) {0x00002, 0xafb29, 0};
45         page_table[2] = (struct tlb_slot_t) {0x00003, 0x4b0dc, 0};
46         page_table[3] = (struct tlb_slot_t) {0x00004, 0x52ca0, 0};
47         page_table[4] = (struct tlb_slot_t) {0x00005, 0xa7cbd, 0};
48         page_table[5] = (struct tlb_slot_t) {0x17d42, 0x338a3, 0};
49         page_table[6] = (struct tlb_slot_t) {0x1238f, 0x28471, 0};
50         page_table[7] = (struct tlb_slot_t) {0xda234, 0x2341b, 0};
51         page_table[8] = (struct tlb_slot_t) {0xf1234, 0x1bca2, 0};
52         page_table[9] = (struct tlb_slot_t) {0x129af, 0x23133, 0};
53
54         /* Init TLB */
55         tlb[0] = page_table[0]; tlb[1] = page_table[1];
56         tlb[2] = page_table[2]; tlb[3] = page_table[3];
57
58         /* Init tests */
59         int test[7] =
60                 {0x00003123, 0x00001524, 0x00002534, 0x17d42e52,
61                 0x121aabdd, 0x000012ac, 0x00004a71};
62
```

```
63          int i;
64          printf("Page table\n");
65          for (i = 0; i < PAGE_TABLE_SIZE; i++) {
66                  printf("%05x --> %05x\n",
67                          page_table[i].virtual, page_table[i].physical);
68          }
69
70          /* Test */
71          printf("Access pages\n");
72          for (i = 0; i < 7; i++) {
73                  address_t addr;
74                  if (translate(test[i], &addr)) {
75                          printf("%08x --> %08x\n", test[i], addr);
76                  }else{
77                          printf("%08x --> Illegal address\n", test[i]);
78                  }
79          }
80
81          /* The TLB */
82          printf("TLB\n");
83          for (i = 0; i < TLB_SIZE; i++) {
84                  printf("%d: %05x --> %05x : %2d\n",
85                          i, tlb[i].virtual, tlb[i].physical, tlb[i].count);
86          }
87
88  }
```

3 Exercise

PROBLEM 1 (5pts). Student need to complete the given source code with "TO DO" part in the sample code. The program aims to simulate the process of translating virtual memory into physical memory with TLB algorithm.

PROBLEM 2 (2pts). Consider the page table shown in Figure 3.1 for a system with 12-bit virtual and physical addresses and with 256-byte pages. The list of free page frames is D, E, F (that is, D is at the head of the list, E is second, and F is last). Convert

Page	Page Frame
0	–
1	2
2	C
3	A
4	–
5	4
6	3
7	–
8	B
9	0

Figure 3.1: Page table for Exercise 1

the following virtual addresses to their equivalent physical addresses in hexadecimal. All numbers are given in hexadecimal. (A dash for a page frame indicates that the page is not in memory.)

- 9EF
- 111
- 700
- 0FF

PROBLEM 3 (3pts). Consider the following page reference string:

$$1, 2, 3, 2, 4, 5, 1, 6, 2, 1, 3, 2, 7, 6, 3, 2, 2, 1, 3, 2, 6.$$

How many page faults would occur for the following replacement algorithms, assuming one, tuo, three, four, five, six and seven frames? Remember that all frames are initiallly empty, so your first unique pages will cost one fault each.

- LRU replacement
- FIFO replacement
- Optimal replacement
- Clock replacement

Draw the graph to illustrate the relationship between number of allocated frames and number of page faults for each replacement algorithms.