EI338 Computer Systems Engineering

Project 8 Designing a Virtual Memory Manager

November 30, 2019

Zhihui Xie, 517030910356

The environment used in this project is **Deepin 15.11**, the latest version of an open source operating system based on Debian's stable branch. The kernel version is **Linux version 4.15.0**.

Exercise

This project consists of writing a program that translates logical to physical addresses for a virtual address space of size $2^16 = 65,536$ bytes. Out program will read from a file containing logical addresses and, using a TLB and a page table, will translate each logical address to its corresponding physical address and output the value of the byte stored at the translated physical address. The learning goal here is to use simulation to understand the steps involved in translating logical to physical addresses. This will include resolving page-faults using demand paging, managing a TLB, and implementing a page-replacement algorithm.

Basic Structure

Our program will read a file containing several 32-bit integer numbers that represent logical addresses. Our program is only concerned with reading logical addresses and translating them into physical addresses. Some specifics include the following:

- 2^8 entries in the page table
- Page size of 2^8 bytes
- 16 entries in the TLB
- Frame size of 2^8 bytes
- 256 frames
- Physical memory of $65,536$ bytes (256 frames $\times$ 256-byte frame size)

Implementation

A visual representation of the address-translation process is shown in **??**. Three main issues should be dealed with:

1. How to extract the page number and offset from logical addresses?
2. How to consult the TLB to obtain the frame number?
3. If the page table must be consulted, how to handle page faults?

Figure 1: Address-translation Process

extract_page_num() & extract_offset()

To address the first issue, we use bit-masking and bit-shifting. **PAGE_MASK** and **OFFSET_MASK** are masking bits.

```
#define PAGE_MASK 0xFF00
#define PAGE_NUM_BIT 8
#define FRAME_NUM_BIT 8
#define OFFSET_MASK 0xFF

// ...

unsigned extract_page_num(unsigned logical_addr) {
    return (logical_addr & PAGE_MASK) >> PAGE_NUM_BIT;
}

unsigned extract_offset(unsigned logical_addr) {
    return logical_addr & OFFSET_MASK;
}
```

Listing 1: extract_page_num() & extract_offset()

0.0.1 get_frame_num_from_TLB()

get_frame_num_from_TLB() is to address the second issue, translating a page number into a frame number by consulting the TLB.

We first check whether the specific page is already in the TLB.

```
for (int i = 0; i < tlb_capacity; ++i) {
    if (tlb[i].page_num == page_num) {
        ++TLB_hit_num;

        return tlb[i].frame_num;
    }
}
```

Listing 2: TLB Hit

If a TLB miss occurs, we call *get_frame_num_from_pt()* to consult the page table, and then update the TLB. If the TLB is not full, we directly add the new page into the TLB. Otherwise, we update it by applying the simplest FIFO policy.

```
// TLB is not full
if (tlb_capacity < TLB_ENTRY_NUM) {
    tlb[tlb_capacity].frame_num = frame_num;
    tlb[tlb_capacity].page_num = page_num;

    ++tlb_capacity;
}

// TLB is full
else {
    tlb[tlb_first_index].frame_num = frame_num;
    tlb[tlb_first_index].page_num = page_num;

    tlb_first_index = (tlb_first_index + 1) % TLB_ENTRY_NUM;
}
```

Listing 3: TLB Miss

get_frame_num_from_pt()

consulting the page table is similar to consulting the TLB. Again, we first try to obtain the frame number by checking whether the corresponding valid bit is set.

```
if (pt[page_num].valid_bit)
    return pt[page_num].frame_num;
```

Listing 4: Consulting the Page Table

If a page fault occurs, we need to find a free-frame in the free frame list. If the page table is full, we select a victim with the FIFO policy and replace it. A new frame is swapped in from **BACKING_STORE**.

```
unsigned frame_num;

// find a free frame
if (free_frame_index < FRAME_NUM) {
    pt[page_num].frame_num = frame_num = free_frame_index++;
    pt[page_num].valid_bit = 1;
    frame_to_page[frame_num] = page_num;
```

```
}

// replace a page
else {
    pt[page_num].frame_num = frame_num = victim_frame_index;
    pt[page_num].valid_bit = 1;
    pt[frame_to_page[frame_num]].valid_bit = 0; // erase
    victim_frame_index = (victim_frame_index + 1) % FRAME_NUM; // circular array
    frame_to_page[frame_num] = page_num;
}

// swap
fseek(backing_store_fp, page_num * PAGE_SIZE, SEEK_SET);
fread(memory + frame_num * FRAME_SIZE, FRAME_SIZE, 1, backing_store_fp);
```

Listing 5: Page Fault

Result

To check whether our program is correct, we conduct an experiment with the test file **address.txt**, which contains 1000 integer values representing logical addresses ranging from 0 to 65535 (the size of the virtual address space). The translation is validated by consulting the ground truth.

Further more, two statistics are to reported:

1. **Page-fault rate**–The percentage of address references that resulted in page faults.
2. **TLB hit rate**–The percentage of address references that were resolved in the TLB .

Our program reports the statistics under two circumstances: **FRAME_NUM** = 128 and **FRAME_NUM** = 256. The results seem reasonable.

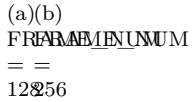

Figure 2: Result