Assignment 4 Walkthrough

Lead TAs: Monum, Shahpar and Mashal

Part 1 - Single Level Paging

Introduction

- Physical Address Space and Virtual Address Space of 2^{16} Bytes

Introduction

- Physical Address Space and Virtual Address Space of 2^{16} Bytes
- Logical Address of 16 bits
- Leftmost 8 significant bit translate to page number
- Rightmost 8 significant bit translate to page offset

Introduction

- Physical Address Space and Virtual Address Space of 2^{16} Bytes
- Logical Address of 16 bits
- Leftmost 8 significant bit translate to page number
- Rightmost 8 significant bit translate to page offset
- Page and Frame size is 2^{8} Bytes

Address Structure

- Address.txt has 32 bit logical addresses
- First left 16 bits will be ignored

32 … 16	15 … 8	7 … 0
Unused	Page No.	Offset

Address Translation

- Translate the logical address to the physical address by using the page table
- When retrieving frame number from page table there are two scenarios
  - It exists in memory and you retrieve the frame number (Yayy less work!)
  - It does not exist and a page fault occurs and you have to retrieve it from the backing store

15 12	11 10	9	8	7 0
Unused	2 bit Counter	D	V/I	Frame No.

Enhanced Second Chance Algorithm

- Same as Second Chance Algorithm but a more enhanced version.
- Two-bit counter instead of one
- (0, 0) neither recently used nor modified—best page to replace
- (0, 1) not recently used but modified—not quite as good, because the
- page will need to be written out before replacement
- (1, 0) recently used but clean—probably will be used again soon
- (1, 1) recently used and modified—probably will be used again soon, and
- the page will be need to be written out to disk before it can be replaced

Handling Page Faults

- There is a backing store provided in a binary file this will emulate as a disk
- Disk size is 65536 Bytes
- When reading a page you will read from the corresponding starting byte to the page till the next 256 bytes.
- Use the following functions to read/write the backing store: fopen(), fread(), fseek(), and fclose() (read the documentation to avoid seg faults)

Getting Started

- addresses.txt contain integers that are logical addresses, you need to convert these logical addresses to physical addresses.
- Convert these integers to binary and mask out the page number and the offset correspondingly.
- Make your own structure for a physical memory and page table (think about how you will make this carefully)

Sample Output

Out.txt:

Logical Address	Physical Address	Read/Write	Value	Page Fault
0xFA1C	0x841C	Read	0x45	Yes

Terminal:

After completion, your program is to report the page-fault rate — the percentage of address references that resulted in page faults (i.e. number of page faults divided by the number of memory addresses read).

Part 2 - Hierarchical Paging

Level Up - Two Level Paging :)

- Address Space is of 2^{24} Bytes hence, logical address is of 24 bits
- Frame size 1024 bytes
- Offset in L1 table: 6 bits, Offset in L2 table: 8 bits, Offset in actual page: 10 bits
- How many entries should L1 Page table and L2 page table have?

31 24	23 18	17 10	9 0
Unused	Offset in L1 Page Table	Offset in L2 Page Table	Offset in Actual Page

The catch :)

- Physical memory is limited to 128 kilobytes. (128 frames)
- You can neither keep all of L2 page table in memory, nor all the frames in the memory.
- Out of 128 frames, 1 frame for the ENTIRE Level 1 page table, 32 frames for Level 2 page table. Rest of the frames to storage pages.
- Level 1 page table will always be in memory.

Handling Page Faults

- Therefore, you will need to use a page replacement strategy.
- You're going to handle page faults similar to part 1 using the Enhanced Second Chance Algorithm.
- Page faults will now occur in both L1 page table and L2 page table :)
- You may store the remainder of the L2 page table in backing store below the last instruction you will be processing. (More on this next)

Page table structure

- 16 bits to store the frame number.
- Leftmost 12 bits will be ignored.

31 20	19 18	17	16	15 0
Unused	2 bit Counter	D	V/I	Frame No.

Program Execution

- You are given BACKING_STORE_2.bin where we have placed a binary form of code. (Recall CS225)
- This binary code starts at (the first instruction is at) address 0x00C17C00 in BACKING_STORE_2.bin and ends at (the last instruction is at) 0x00C193E8.
- **Your task is to extract this binary from BACKING_STORE_2.bin and execute this binary code using your 2- level paging system.**

Decoding binary code

- Each instruction is of 8 bytes and each memory address is of 3 bytes.
- Two types of instructions: memory-memory instructions, memory-value instructions. (Again recall CS225)
- **Memory-memory** instruction: both arguments are memory addresses, read value from both addresses, and store result in the first memory address.
- **Memory-value** instruction: first argument is a memory address and second argument is an immediate value. The result is stored at memory address.

Instruction structure

Memory-Memory Instruction Structure:

Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0
Opcode	First Address			Second Address			Unused

Memory Value Instruction Structure:

Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0
Opcode	First Address			Value			

Example

- For example, if the instruction is 0x601011e211111110, the opcode 60 indicates that the instruction is **OR** and it is a memory-value instruction.
- To execute this instruction, you need to read the value at address **0x1011e2** from BACKING_STORE_2.bin let's call the value **A**.
- Take bitwise OR of **A with 0x11111110** and store the result of the operation at address **0x1011e2.**

Expected Output

For each instruction:

- Type of instruction.
- Whether the memory address access was a page hit or miss for both inner and outer page table.
- The value at that address.
- Check handout for detailed format.

General advice for both parts

- Start with declaring Global constants (MEM_SIZE, DIRTY_BIT_MASK, NUM_FRAMES, PAGE_SIZE, etc)
- Start with writing the pseudocode and the tasks you need to perform.
- Modular Design: Divide each task in functions. Test each function on the go. This will save you from a lot of trouble later. Debugging the whole code in the end will be a mess, do not that to yourself.
- START EARLY!!!!!!!!!!! You CANNOT complete this in the last two days.