Project 3

Virtual Memory Manager

Alex Benasutti and Alex Van Orden

CSC345-01

November 8 2019

C Code (main.c):

```
/*
* Alex Benasutti and Alex Van Orden
* CSC345-01
* November 8 2019
* Project 3 - Virtual Memory Manager
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdint.h>
#include <time.h>

#define MAX_PT_ENTRIES 256      /* Maximum page table entries */
#define PAGE_SIZE 256           /* Page size of 256 bytes */
#define MAX_TLB_ENTRIES 16      /* Maximum TLB entries */
#define FRAME_SIZE 256          /* Frame size of 256 bytes */
#define MAX_FRAMES 256          /* Maximum frames in physical memory */

#define BYTES_PER_INPUT 256     /* Chunk of bytes to read from backing_store */

typedef struct
{
    int page_num;
    int frame_num;
} page_frame;

int physical_memory[MAX_FRAMES][FRAME_SIZE]; /* Physical memory holds 256 bytes per frame */
page_frame TLB[MAX_TLB_ENTRIES];             /* TLB holds a page number and a corresponding
frame number */
page_frame page_table[MAX_PT_ENTRIES];       /* Page table holds a page number and a
corresponding frame number */

int page_faults = 0;    /* Page fault counter */
int tlb_hits = 0;       /* TLB hit counter */
int framePos = 0;       /* Next available frame position in physical memory */
int PTPos = 0;          /* Next available position in page table */
int TLBPos = 0;         /* Next available position in TLB */
int addressCount = 0;   /* Translated address counter */

/* Load BACKING_STORE.bin - mirror of everything in logical address space */
FILE *backing_store;
/* Create output files for virtual addresses, physical addresses, and values */
FILE *virtual, *physical, *val;

void getPage(int address);
void readStore(int page_num);
void TLBInsert(int page_num, int frame_num);

int main(int argc, char** argv)
{
    /* Load addresses through input file */
    FILE *input;
    if (argc == 2)
    {
        char const* const fileName = argv[1];
```

```
        input = fopen(fileName, "rt");

        /* Open output files */
        virtual = fopen("out1.txt","w");
        physical = fopen("out2.txt","w");
        val = fopen("out3.txt","w");
    }
    else
    {
        fprintf(stderr,"Input address file not specified.\n");
        return -1;
    }

    /* Declare logical address variables */
    int32_t logical_address;
    fscanf(input, "%d", &logical_address);

    /* Extract each logical address from the address file */
    while (!feof(input))
    {
        addressCount++;
        /* Get physical address and valued stored at that address */
        getPage(logical_address);

        /* Scan for any new integers on proceeding lines */
        fscanf(input, "%d", &logical_address);
    }
    printf("%d addresses.\n", addressCount);
    fclose(input);
    fclose(virtual);
    fclose(physical);
    fclose(val);

    double pf_rate = page_faults / (double)addressCount;
    double tlb_rate = tlb_hits / (double)addressCount;

    /* Will not run out of memory with 256 frames */
    printf("Page Faults: %d\tPage Fault Rate: %.2f\n", page_faults, pf_rate);
    printf("TLB Hits: %d\t TLB Rate: %.3f\n", tlb_hits, tlb_rate);

    return 0;
}

void getPage(int logical_address)
{
    /* Declare page (0x0000FF00) and offset (0x000000FF) */
    uint8_t page = logical_address >> 8 & 0xFF; // unsure whether these should be ints or not
    uint8_t offset = logical_address & 0xFF;
    int frame = -1;
    int i;

    /* Search TLB for page and set frame number if found */
    for (i=0;i<TLBPos;i++)
    {
        if (TLB[i].page_num == page)
        {
            frame = TLB[i].frame_num;
```

```
            tlb_hits++;
            break;
        }
    }

    /* If not found, search page table and set frame number if found */
    if (frame == -1)
    {
        for (i=0;i<PTPos;i++)
        {
            if (page_table[i].page_num == page)
            {
                /* Page was found - get frame */
                frame = page_table[i].frame_num;
                break;
            }
        }
    }

    /* If not found, declare page fault and read in frame number from backing_store
(readFromStore) */
    /* Update page table entry, restart address conversion and access procedure */
    if (frame == -1)
    {
        readStore(page);
        page_faults++;
        /* Decrement updated framePos to get frame of current page */
        frame = framePos - 1;
    }

    /* Insert page num and frame into TLB if not there already */
    TLBInsert(page, frame);

    /* logical_address -> out1.txt */
    fprintf(virtual,"%d\n",logical_address);
    /* frame_num * 256 + offset -> out2.txt */
    int physical_address = frame * PAGE_SIZE + offset;
    fprintf(physical,"%d\n",physical_address);
    /* Get signed byte value stored in physical_memory using frame_num and offset -> out3.txt */
    int8_t value = physical_memory[frame][offset];
    fprintf(val,"%d\n",value);
}

void readStore(int page_num)
{
    /* Seek for page_num in backing_store */
    /* Bring corresponding frame num to physical_memory and page_table */

    /* Declare buffer to hold 256 bytes */
    int8_t buf[256];
    int i;

    /* Open BACKING_STORE.bin */
    backing_store = fopen("BACKING_STORE.bin","rb");

    /* Set file pointer at page number position */
    if (fseek(backing_store, page_num*BYTES_PER_INPUT, SEEK_SET) != 0)
```

```
    {
        fprintf(stderr,"Error seeking through backing_store\n");
    }

    /* Read 256 bytes to buffer array */
    if (fread(buf, sizeof(int8_t), BYTES_PER_INPUT, backing_store) == 0)
    {
        fprintf(stderr, "Error reading through file\n");
    }

    /* Close backing_store */
    fclose(backing_store);

    /* Physical memory has not filled */
    if (framePos < MAX_FRAMES)
    {
        /* Update frame in physical memory with 256 bytes */
        for (i=0;i<BYTES_PER_INPUT;++i) physical_memory[framePos][i] = buf[i];

        /* Update page table with page and frame */
        page_table[PTPos].page_num = page_num;
        page_table[PTPos].frame_num = framePos;

        /* Increment frame and page table's next available position */
        framePos++;
        PTPos++;
    }
    else /* Physical memory/PT is filled - use FIFO for replacement */
    {
        int j;
        for (j=0;j<MAX_FRAMES-1;j++)
        {
            /* Push top mem out of queue */
            for (i=0;i<BYTES_PER_INPUT;++i) physical_memory[j][i] = physical_memory[j+1][i];
            /* Push top page out of queue */
            page_table[j] = page_table[j+1];
        }

        /* Add new memory to back of physical memory */
        for (i=0;i<BYTES_PER_INPUT;++i) physical_memory[j][i] = buf[i];

        /* Add new page and frame to back of page table */
        page_table[j].page_num = page_num;
        page_table[j].frame_num = framePos-1;
    }
}

void TLBInsert(int page_num, int frame_num)
{
    /* Insert page and frame into TLB */
    int i, j;

    /* Break if already on the TLB and save index */
    for (i=0;i<TLBPos;i++)
    {
        if (TLB[i].page_num == page_num) break;
    }
```

```
/* If page number was not found in TLB */
if (i == TLBPos)
{
    /* Insert to TLB if there is still room */
    if (TLBPos < MAX_TLB_ENTRIES)
    {
        TLB[TLBPos].page_num = page_num;
        TLB[TLBPos].frame_num = frame_num;
    }
    else /* Otherwise move everything over and insert page frame on end */
    {
        for (i=0;i<MAX_TLB_ENTRIES-1;i++)
        {
            TLB[i] = TLB[i+1];
        }
        /* Insert new page frame */
        TLB[MAX_TLB_ENTRIES-1].page_num = page_num;
        TLB[MAX_TLB_ENTRIES-1].frame_num = frame_num;
    }
}
else /* If page num was found in TLB */
{
    /* Increment position of everything starting from page index */
    for (i=i;i<MAX_TLB_ENTRIES-1;i++)
    {
        TLB[i] = TLB[i+1];
    }
    /* If there is still room, put page and frame on the end */
    if (TLBPos < MAX_TLB_ENTRIES)
    {
        TLB[TLBPos].page_num = page_num;
        TLB[TLBPos].frame_num = frame_num;
    }
    else /* Otherwise, place page and frame on num entries - 1 */
    {
        TLB[TLBPos-1].page_num = page_num;
        TLB[TLBPos-1].frame_num = frame_num;
    }
}
if (TLBPos < MAX_TLB_ENTRIES) TLBPos++;

// printf("TLB INSERT POSITION: %d - ", TLBPos);
/* TLB has not filled */
// if (TLBPos < MAX_TLB_ENTRIES)
// {
//     // printf("ADDING TO TLB\n");
//     TLB[TLBPos].page_num = page_num;
//     TLB[TLBPos].frame_num = frame_num;
//     TLBPos++;
// }
// else /* TLB is filled - use FIFO for page replacement */
// {
//     // printf("TLB FILLED - REPLACE ENTRY\n");
//     for (j=0;j<MAX_TLB_ENTRIES-1;j++)
//     {
//         TLB[j] = TLB[j+1];                  /* Push top out of queue */
```

```
    //      }
    //      TLB[j].page_num = page_num;             /* Place new val in bottom of stack */
    //      TLB[j].frame_num = frame_num;
    // }
}
```

Implementation

Highlighted in **red** is the implementation for reading in all input logical addresses. By utilizing the fscanf() function, each line of "addresses.txt" can be read and saved into the program as an integer and parsed through the getPage() function for further operations. Some of these operations include masking the logical address to find *page* and *offset* variables, and outputting the logical address to "out1.txt". Using a feof() check, the program will stop reading logical addresses once it has reached the end of the input file.

Highlighted in **blue** shows the code necessary for translating a logical address to a physical address. The equation used for calculating the physical address is *frame* * *page size* + *offset*. Page size is defined as a constant by the program/write-up as a 256 byte value. Offset, as stated above, is found by masking the last 8 bits of the parsed logical address. The frame number is the specific frame holding the page number parsed from the logical address, and this can be found through a number of means. The program first checks the TLB for the page, and if found, returns its associated frame number. If not in the TLB, the program will then check each entry of the page table for the page. If the frame still cannot be found, the program must load the page number from the "BACKING_STORE.bin" binary file, and set a new frame number. Once all three variables are obtained, the physical address is calculated and outputted to "out2.txt".

Highlighted in **orange** is the code that correctly retrieves the values stored in the physical addresses. This value is retrieved from the *physical_memory[][]* variable at the logical address page's corresponding frame. Each frame in the physical memory carries 256 bytes of data, so the program needs a specific byte to index for. The byte value that the program index is at the logical address' *offset* value. Once this value is retrieved, the program outputs it to "out3.txt".

Highlighted in **purple** is the FIFO-based TLB update for the VMM. Immediately after a frame number is declared, the TLBInsert() procedure is called to add both the page number and frame number to the TLB array. The program first checks for whether the page number is already present in the TLB. If not, the new page-frame is added to the TLB. Although, because the TLB only contains room for 16 entries, a replacement algorithm must be used to make room for new page-frames. If the TLB is full, a FIFO-based approach is used and the program replaces the oldest page-frame, which always sits at the front of the TLB.

A special case occurs when the page-frame in question is already present in the TLB. For the implementation without page-replacement, and LRU-based algorithm is used and moves the position of the present page-frame to the back of the TLB.

Commented out under the TLB update code is the FIFO-based algorithm used for the TLB update of the page-replacement VMM.

Highlighted in **magenta** shows the implementation for calculating the correct number of page faults in the VMM. A global page fault counter is incremented when a frame number corresponding to the logical address' page number cannot be located within either the TLB or the page table. In tandem with this, the program executes the readStore() procedure, which reads in a 256-byte page from "BACKING_STORE.bin" and stores it into an available page-frame in physical memory. This (hopefully) assures that the next time this page is called, it is stored within either the TLB or the page table and a page fault will not occur. Before the program exists, it will report the page fault count and ratio to the amount of translated logical addresses.

Highlighted in **dark red** shows the counting block for TLB hits. The very first check for acquiring a page-frame is through the 16-entry TLB. If the page-frame can be found stored in the TLB, the *frame* variable will be set and the program will not have to search through the page table or BACKING_STORE. Additionally, the program will increment a global TLB hit counter. Before the program exits, it will report the TLB hit count and ratio to the amount of translated logical addresses.

Highlighted in **green** is the implementation of a FIFO-based page replacement algorithm. So long as the physical memory and page table have not filled, the program will have no problem adding new page-frames. However, once they have reached capacity, the program will employ a FIFO-based page replacement algorithm, pushing out the oldest memory/page-frame while adding new data to the back of the physical memory/page table. While the code shown above is for main.c and not main_pr.c, this algorithm could be applied to both programs considering that this won't cause an error in main.c.