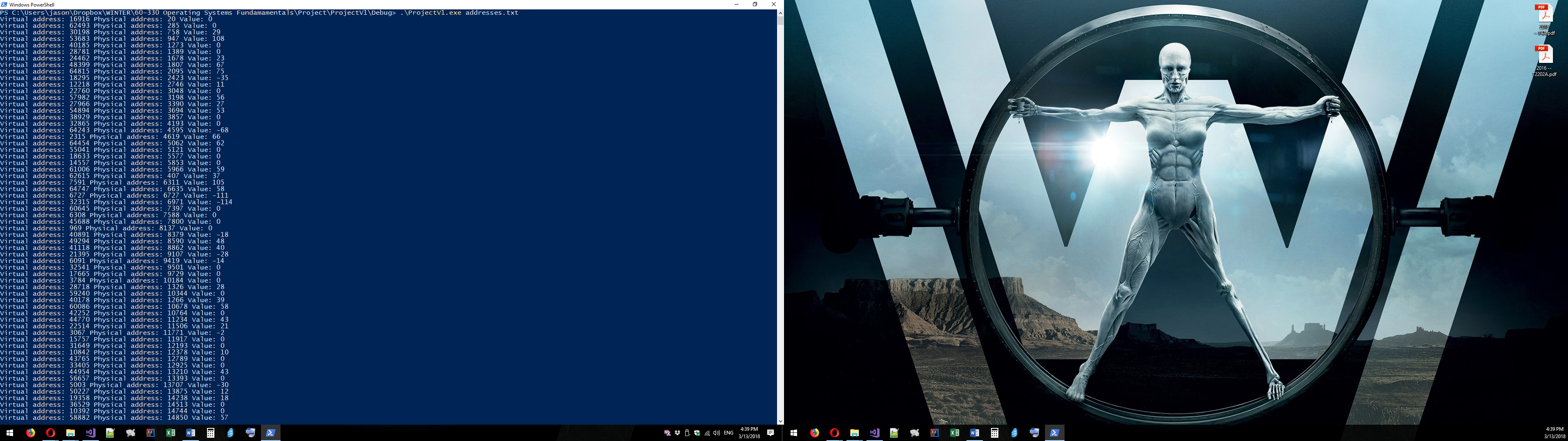
*I confirm that I will keep the content of this project confidential. I confirm that I have not received any unauthorized assistance in preparing for or writing this project. I acknowledge that a mark of 0 may be assigned for copied/plagiarized work:* Jason Choquette 104 337 378.

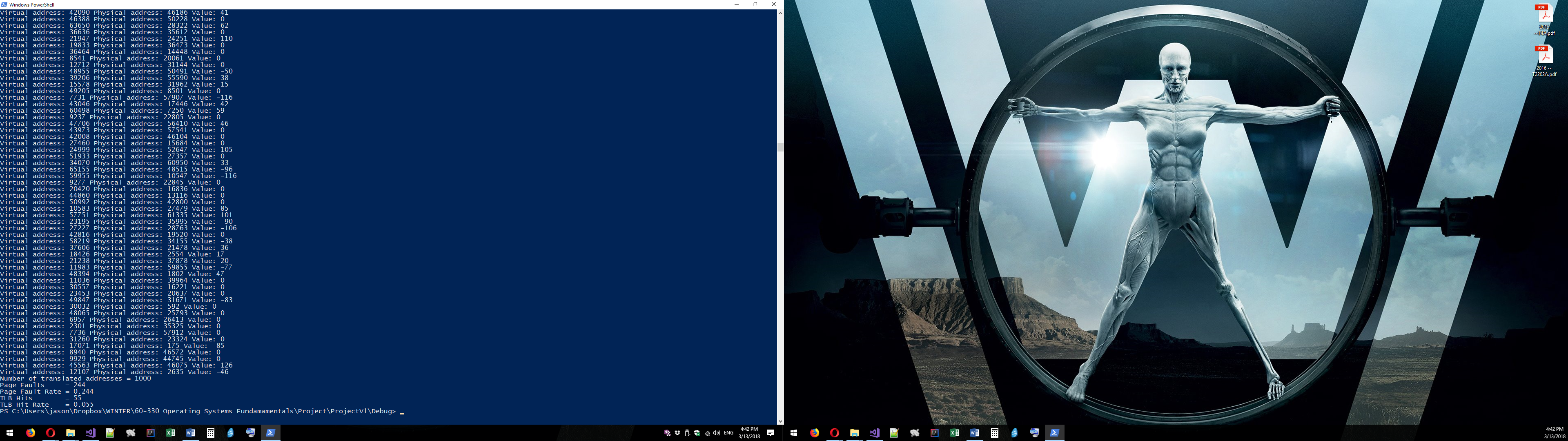
60-315 Winter 2018 Course Project

Sample run of program:

Beginning output:



Ending output:



My tlb hits are off by 1 (compared to correct.txt). Not sure why? I tried to figure it out but couldn’t see what I did wrong?

Since the logical addresses were random, the tlb hit rate is rather low. In reality, during program execution, the CPU will be reading instructions and data where the addresses are within a limited range (perhaps as low as 4096 kb for most stacks). Therefore, more frames and pages will be located on the tlb and the hit rates will be higher. Complier optimization will also help to create a “tighter” instruction and data variability. Since this was a purely academic exercise with a software-implemented tlb, the effects were not seen in the statistics.

Program analysis

I attempted the two-person project since the single person project was fairly simple. The design of the program is straight-forward:

The main function reads in the addresses.txt file, validates the input and then calls a function called process\_addresses() (function prototypes and global variables and constants will be provided at the end of the report.).

Once the call returns, the statistics of the program and address translations are printed to console.

Below is the main function, process\_addresses function and print\_main\_results function:

int main(const int argc, char \*argv[])

{

char address[MAX\_BUFFER\_SIZE];

backing\_store\_name = "BACKING\_STORE.bin";

int number\_of\_translated\_addresses = 0;

tlb\_hits = 0;

if (argc != 2)

{

fprintf(stderr,"Usage: ./a.out [input file]\n");

return -1;

}

if(!valid\_file\_input(backing\_store = fopen(backing\_store\_name, "rb"),

backing\_store\_name)) return -1;

if(!valid\_file\_input(address\_file  = fopen(argv[1], "r"),

argv[1])) return -1;

process\_addresses(address, address\_file, &number\_of\_translated\_addresses);

print\_main\_results(number\_of\_translated\_addresses);

// close the input file and backing store

fclose(address\_file);

fclose(backing\_store);

return 0;

}

void process\_addresses(FILE \*address\_file, int \* number\_of\_translated\_addresses)

{

char address[MAX\_BUFFER\_SIZE];

// read through the input file and output each logical address

while ( fgets(address, MAX\_BUFFER\_SIZE, address\_file) != NULL)

{

// get the physical address and value stored at that address

find\_page(atoi(address));

\*number\_of\_translated\_addresses += 1;

}

}

void print\_main\_results(const int number\_of\_translated\_addresses)

{

// calculate and print out the stats

printf("Number of translated addresses = %d\n", number\_of\_translated\_addresses);

const double pf\_rate  = page\_faults / (double)number\_of\_translated\_addresses;

const double tlb\_rate = tlb\_hits    / (double)number\_of\_translated\_addresses;

printf("Page Faults     = %d\n",   page\_faults);

printf("Page Fault Rate = %.3f\n", pf\_rate);

printf("TLB Hits        = %d\n",   tlb\_hits);

printf("TLB Hit Rate    = %.3f\n", tlb\_rate);

}

The print\_main\_results() function is straight-forward so I will discuss the implementation of the process\_addresses() function.

The function reads through the addresses.txt file and for each logical address in the file, a function called find page is called, and the number of addresses counter is incremented. The find\_page() function searches the tlb and page table for a page. If the page is not found in either of those data structures, a backing store (long-term storage) is searched for on-demand-paging.

We will look at the find\_page() function next:

bool find\_page(const int logical\_address)

{

// obtain the page number and offset from the logical address

const int page\_number  = ((logical\_address & ADDRESS\_MASK) >> 8);

const int offset       =  (logical\_address & OFFSET\_MASK);

bool page\_fault\_raised = FALSE;

bool page\_found        = FALSE;

int frame\_number       = -1;

// first try to get page from TLB

search\_tlb(page\_number, &page\_found, &frame\_number);

// if the frame number was not found in tlb, search the page\_table...

if(!page\_found)

search\_page\_table(page\_number, &frame\_number, &page\_fault\_raised);

if(page\_fault\_raised)

{

read\_from\_store(page\_number);           frame\_number = first\_available\_frame - 1; // set the frameNumber to the current fi rstAvailableFrame index

tlb\_insert(page\_number, frame\_number);

}

// frame number and offset used to get the signed value stored at that address

const signed char value = physical\_memory[frame\_number][offset];

printf("Virtual address: %d Physical address: %d Value: %d\n", logical\_address, (f rame\_number << 8) | offset, value);

return page\_fault\_raised;

}

This function first obtains the page number and offset from the logical address and then sets some flags. Once those are completed, we search the simulated tlb for the page number. If the page is not found, search the page table.

Again, if the page is not found in the page table, a page fault is raised and a search is made in the backing store.

Through one of these functions, the page will be found and the tlb will be updated. Once updated the result is printed to the console.

This process is repeated for each logical address in the file.

Next, we look at the search functions:

* Search\_tlb
* Search\_page\_table

void search\_tlb(const int page\_number, bool \* page\_found, int \* frame\_number)

{

for (int i = 0; i < TLB\_SIZE; i++)

if(tlb\_page\_number[i] == page\_number)

{

\*frame\_number = tlb\_frame\_number[i];

tlb\_hits++;

\*page\_found = TRUE;

}

}

void search\_page\_table(const int page\_number, int \* frame\_number, bool \* page\_fault\_raised)

{

for (int i = 0; i < first\_available\_page\_table\_number; i++)

if(page\_table\_numbers[i] == page\_number)

// if the page is found in those contents

\*frame\_number = page\_table\_frames[i]; // extract the frame number

if(\*frame\_number == -1)

{

\*page\_fault\_raised = TRUE;

page\_faults++;

}

}

These functions loop through each data structure until a match is found. For the search\_page\_table(), if the page is not found the page fault is raised.

After these function calls, a check is made to see a page fault was raised and if it was, a search is made in the backing store.

if(page\_fault\_raised)

{

read\_from\_store(page\_number);

frame\_number = first\_available\_frame - 1;

tlb\_insert(page\_number, frame\_number);

}

Here, we read from the store and then insert the page number and frame number into the tlb.

We look at these functions next.

void read\_from\_store(const int page\_number)

{

// the buffer containing reads from backing store

signed char buffer[PAGE\_SIZE];

// first seek to byte PAGE\_SIZE in the backing store

// SEEK\_SET in fseek() seeks from the beginning of the file

if ( fseek(backing\_store, page\_number \* PAGE\_SIZE, SEEK\_SET) != 0 )

fprintf(stderr, "Error seeking in backing store\n");

// now read PAGE\_SIZE bytes from the backing store to the buffer

if ( fread(buffer, sizeof(signed char), PAGE\_SIZE, backing\_store) == 0)

fprintf(stderr, "Error reading from backing store\n");

for(int i = 0; i < PAGE\_SIZE; i++)

physical\_memory[first\_available\_frame][i] = buffer[i];

// and then load the frame number into the page table in the first available frame

page\_table\_numbers[first\_available\_page\_table\_number] = page\_number;

page\_table\_frames[first\_available\_page\_table\_number] = first\_available\_frame;

// increment the counters that track the next available frames

first\_available\_frame++;

first\_available\_page\_table\_number++;

}

Since we know what page we are looking for we directly find the page in the backing store using fseek(). Then we store address into the simulated physical memory.

Next is the function tlb\_insert().

The tlb\_insert() function uses a FIFO design to replace the last element of the array which has been shifted on each tlb insert from beginning to end. So, the last element is the oldest.

See code on next page.

void tlb\_insert(const int page\_number, const int frame\_number)

{

int i;

// if it's already in the TLB, break

for (i = 0; i < number\_of\_tlb\_entries; i++)

if(tlb\_page\_number[i] == page\_number)

break;

// if the number of entries is equal to the index

if (i == number\_of\_tlb\_entries)

{

if (number\_of\_tlb\_entries < TLB\_SIZE)

{

// the TLB still has room in it

tlb\_page\_number[number\_of\_tlb\_entries]  = page\_number;     // insert the page and frame onto the end of the array

tlb\_frame\_number[number\_of\_tlb\_entries] = frame\_number;

}

// otherwise move everything over by 1

else

{

for (i = 0; i < TLB\_SIZE - 1; i++)

{

tlb\_page\_number[i]  = tlb\_page\_number[i + 1];

tlb\_frame\_number[i] = tlb\_frame\_number[i + 1];

}

// and insert the page and frame on the end

tlb\_page\_number[number\_of\_tlb\_entries-1]  = page\_number;

tlb\_frame\_number[number\_of\_tlb\_entries-1] = frame\_number;

}

}

else

{

// iterate through up to one less than the number of entries

for (; i < number\_of\_tlb\_entries - 1; i++)

{

// move everything over in the arrays

tlb\_page\_number[i]  = tlb\_page\_number[i + 1];

tlb\_frame\_number[i] = tlb\_frame\_number[i + 1];

}

// if there is still room in the array, put the page and frame on the end

if (number\_of\_tlb\_entries < TLB\_SIZE)

{

tlb\_page\_number[number\_of\_tlb\_entries]  = page\_number;

tlb\_frame\_number[number\_of\_tlb\_entries] = frame\_number;

}

// otherwise put the page and frame on the number of entries - 1

else

{

tlb\_page\_number[number\_of\_tlb\_entries-1]  = page\_number;

tlb\_frame\_number[number\_of\_tlb\_entries-1] = frame\_number;

}

}

// if there is still room in the arrays, increment the number of entries

if (number\_of\_tlb\_entries < TLB\_SIZE)

number\_of\_tlb\_entries++;

}

PROJECT PROTOTYPES, GLOBALS AND CONSTANTS

#ifndef \_PROTOTYPES\_H\_

#define \_PROTOTYPES\_H\_

#include <stdio.h>

#include <stdlib.h> // atoi

// Project Constants

#define FRAME\_SIZE 256 // size of the frame

#define TOTAL\_NUMBER\_OF\_FRAMES 256 // total number of frames in physical memory

#define ADDRESS\_MASK 0xFFFF // mask all but the address

#define OFFSET\_MASK 0xFF // mask all but the offset

#define TLB\_SIZE 16 // size of the TLB

#define PAGE\_SIZE 256 // size of the page table

#define MAX\_BUFFER\_SIZE 10 // max chars from input file

// global arrays

int physical\_memory[TOTAL\_NUMBER\_OF\_FRAMES][FRAME\_SIZE];

int tlb\_page\_number[TLB\_SIZE];

int tlb\_frame\_number[TLB\_SIZE];

int page\_table\_numbers[PAGE\_SIZE];

int page\_table\_frames[PAGE\_SIZE];

// global variables

int page\_faults;

int tlb\_hits;

int first\_available\_frame;

int first\_available\_page\_table\_number;

int number\_of\_tlb\_entries;

// global file pointers

const char \* backing\_store\_name;

FILE       \* backing\_store;

FILE    \* address\_file;

typedef enum { FALSE, TRUE } bool;

// function prototypes

bool find\_page(const int logical\_address);

void read\_from\_store(const int page\_number);

void tlb\_insert(const int page\_number, const int frame\_number);

void search\_tlb(const int page\_number, bool \* page\_found, int \* frame\_number);

void search\_page\_table(const int page\_number, int \* frame\_number, bool \* page\_fault\_raised);

void print\_main\_results(int number\_of\_translated\_addresses);

void process\_addresses(FILE \* address\_file, int \* number\_of\_translated\_addresses);

bool valid\_file\_input(FILE \* file, const char \* file\_name);

#endif \_PROTOTYPES\_H\_