Course: ECE 408 (Operating Systems)

Name: Josh Martin

HMW: 5

Date: 04/11/2019

Introduction

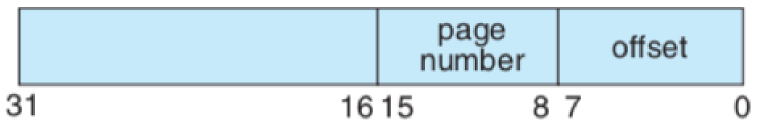
This project consists of writing a program that translates logical addresses to physical addresses, for a virtual address space 2^{16}. My program reads from a file containing logical addresses and, using a TLB as well as a page table, will translate each logical address to its corresponding physical address and output the value of the byte stored at the translated physical address. The goal behind this project is to simulate the steps involved in translating logical to physical addresses. This project assumes that the physical memory is the same size as the virtual address space.

Methodology

The program reads a file containing several 32-bit integer numbers that represent logical addresses. However, It is only important to consist rightmost 16-bits. This 16-bits is divided into:

1. 8-bits page number
2. 8-bit page offset

Address Structure



- Physical Memory is broken into fixed-sized blocks called frame
- Logical memory is broken into blocks of equal size blocks called pages.
- Every address generated by the CPU is divided into two parts:
  - Page Number, used to index into the page table
  - Page Offset
- 2^8 entries in page table

- Page size of 2^8 bytes
- 16 entries in the TLB
- Frame size of 2^8 bytes
- 256 frames
- Physical memory of 2^{16}

```
#define FRAME_SIZE 256
#define TOTAL_NUMBER_OF_FRAMES 256
#define TLB_SIZE 16
#define PAGE_TABLE_SIZE 256
#define ADDRESS_BUFFER_SIZE  10

typedef struct page_data {
    int TableNumbers[PAGE_TABLE_SIZE];//holds the page numbers
    int TableFrames[PAGE_TABLE_SIZE]; //holds the frame numbers
    int faults;// counts page faults
} Page;

typedef struct TLB_data {
    int pageNumber[TLB_SIZE];
    int frameNumber[TLB_SIZE];
    int hits;//counts TLB hits
    int entries;//counts the number of entries in the TLB
} TLB_data;

typedef struct data_struct {
    FILE *address_file;
    FILE *backing_store;
    Page pager;
    TLB_data TLB;
    int physicalMemory[TOTAL_NUMBER_OF_FRAMES][FRAME_SIZE];
    int firstAvailableFrame; //tracks the first available frame
    int firstAvailablePageTableNumber; //tracks the first available page table entry
} virtual_memory;
```

Creation of Addresses

Address are read in from the `address.txt` file into the address array. Which then gets converted into an integer then used in the `getPage` to create the TLB entry.

The program will keep cycling through addresses until `address.txt` is empty.

```
# main()
# ...
    char address[ADDRESS_BUFFER_SIZE];
```

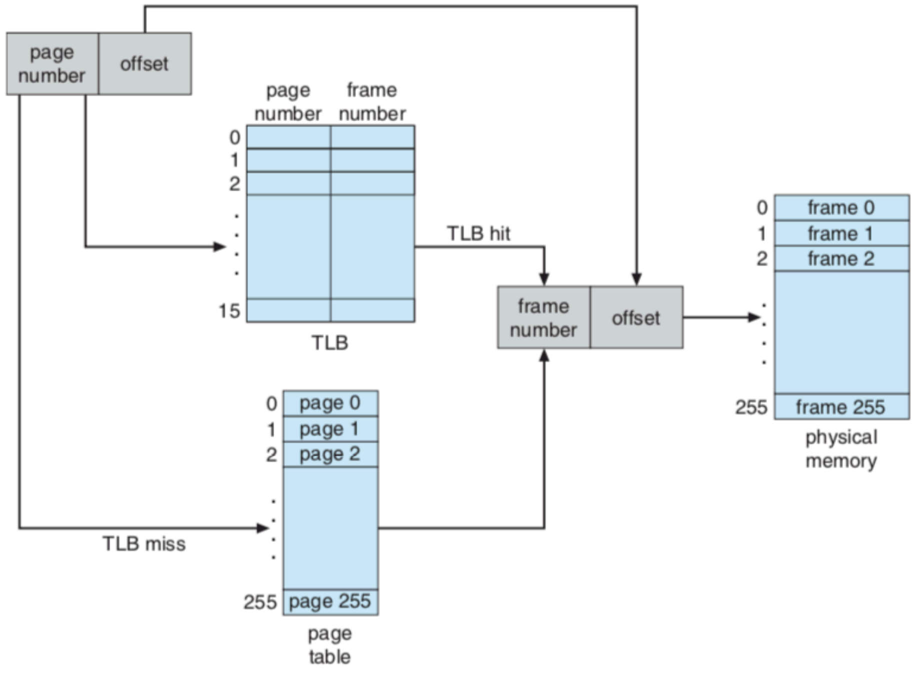
```
# ..
    _this->address_file = openFile(argv[1], "r");

    while (fgets(address, ADDRESS_BUFFER_SIZE, _this->address_file) != NULL ) {
        getPage(_this, atoi(address)); // get the physical address and value
        addresses_seen++;  // increment the number of translated addresses
    }

    fclose(_this->address_file);
```

Address Translation

If the page number is not in the TLB, then a TLB miss happens. The memory reference to the page table must be made. When the frame number is obtained, it is now possible to access memory. In the next access, this chunk of memory will be more easily accessible by the use of it's page number and frame number in the TLB.



Creating the Page Number and Offset

In `getPage` the logical address are read in as 32bit address. To get the Page number, we logical AND the logical address with `0xFF00` to ignore the last 8-bits and to get the correct Page number, we need to shift everything to the right by 1 byte. To obtain the Offset we will apply logical AND with `0x00FF` such that it will discard the left most byte.

```
# getPage( virtual_memory * _this, int logical_address)

    int pageNumber = ((logical_address & 0xFF00)>>8); // 8-bit page number
    int offset = (logical_address & 0x00FF); // 8-bit page offset
```

Determining the frame number

```
# getPage( virtual_memory * _this, int logical_address)
    int frame_number = find_frame(_this, pageNumber);
```

If the Page number is in the TLB, set current index's TLB's frame number to the `frame_number` and increment `TlB.hits` for statical purpose.

```
#find_frame(virtual_memory * _this, int pageNumber)
    for(i = 0; i < TLB_SIZE; i++){
        if(_this->TLB.pageNumber[i] == pageNumber){
            frame_number = _this->TLB.frameNumber[i];
            _this->TLB.hits++;
        }
    }
```

If the Page Number is not in the TLB, then we search the page table to assert if an entry exist in it. `frame_number` is then assigned the frame in the page table.

```
    if(frame_number == -1){ // frame_number not found
        for(i = 0; i < _this->firstAvailablePageTableNumber; i++){
            if(_this->pager.TableNumbers[i] == pageNumber){
                frame_number = _this->pager.TableFrames[i];
            }
        }
```

Handling Page Faults

If the Page Number does not exist in either the TLB or page table, then we count it as **page fault**

```
#find_frame(virtual_memory * _this, int pageNumber)
        if(frame_number == -1){// the page is not found in those contents
            getStore(_this, pageNumber); // gets data from .bin
            _this->pager.faults++;
            frame_number = _this->firstAvailableFrame - 1;
        }
```

```
    }
    return frame_number;
```

The disk space data is represented by the file `BACKING_STORE.bin`. When a page fault occurs, the program reads a 256 byte chuck from the BACKING STORE and stores it to an available space in memory. The `BACKING_STORE.bin` is treated as random access.

```
#define BUFFER_SIZE  256 // number of bytes to read

#getStore(virtual_memory *_this, int pageNumber)

    signed char buffer[BUFFER_SIZE];

    if (fseek(_this->backing_store, pageNumber * BUFFER_SIZE, SEEK_SET) != 0) {
        fprintf(stderr, "Error seeking in backing store\n");
    }

    // now read BUFFER_SIZE bytes from the backing store to the buffer
    if (fread(buffer, sizeof(signed char), BUFFER_SIZE, _this->backing_store) == 0) {
        fprintf(stderr, "Error reading from backing store\n");
    }

    // load the bits into the first available frame in the physical memory 2D array
    for(int i = 0; i < BUFFER_SIZE; i++){
        _this->physicalMemory[_this->firstAvailableFrame][i] = buffer[i];
    }

    // load the frame number into the page table in the first available frame
    _this->pager.TableNumbers[_this->firstAvailablePageTableNumber]
    = pageNumber;

    _this->pager.TableFrames[_this->firstAvailablePageTableNumber]
    = _this->firstAvailableFrame;

    //track the next available frames
    _this->firstAvailableFrame++;
    _this->firstAvailablePageTableNumber++;
```

Creating TLB and FIFO page replacement

Since we now have the `pageNumber` and `frame_number`, we can now queue TLB to assert if an entry exist. If not, we can create a new one into the TLB.

```
# getPage( virtual_memory * _this, int logical_address)
    setIntoTLB(_this, pageNumber, frame_number);
```

We first check if an entry exist in the TLB

```
# setIntoTLB(virtual_memory *_this, int pageNumber, int frameNumber)
    int i;  // if it's already in the TLB, break
    for(i = 0; i < _this->TLB.entries; i++){
        if(_this->TLB.pageNumber[i] == pageNumber){
            break;
        }
    }
```

If `i == _this->TLB.entries` then means an entry exist in the TLB. We then need to check if the TLB is full. If the TLB is not full then we can insert at the end of the TLB. Otherwise, we will data out of the TLB and insert our new data.

```
    if(i == _this->TLB.entries){
        if(_this->TLB.entries < TLB_SIZE){
            // insert the page and frame on the end
            _this->TLB.pageNumber[_this->TLB.entries] = pageNumber;
            _this->TLB.frameNumber[_this->TLB.entries] = frameNumber;
        }
        else{// otherwise shift everything
            for(i = 0; i < TLB_SIZE - 1; i++){
                _this->TLB.pageNumber[i] = _this->TLB.pageNumber[i + 1];
                _this->TLB.frameNumber[i] = _this->TLB.frameNumber[i + 1];
            }
            _this->TLB.pageNumber[_this->TLB.entries-1] = pageNumber;
            _this->TLB.frameNumber[_this->TLB.entries-1] = frameNumber;
        }
    }
```

This process above and below is a FIFO page replacement. FIFO page replacement is simplest page replacement algorithm replacing based on when each page was enter into the TLB and replacing the oldest page if the TLB is full.

If entry exist but not at the end of the TLB then we shift everything from the current index forward.

```
    else{ // index is not <==> to # of entries
        for(; i < _this->TLB.entries - 1; i++){
            _this->TLB.pageNumber[i] = _this->TLB.pageNumber[i + 1];
            _this->TLB.frameNumber[i] = _this->TLB.frameNumber[i + 1];
        }
        if(_this->TLB.entries < TLB_SIZE){// if there is room, put @ end
            _this->TLB.pageNumber[_this->TLB.entries] = pageNumber;
            _this->TLB.frameNumber[_this->TLB.entries] = frameNumber;
        }
        else{// put the page and frame @  (entries - 1)
            _this->TLB.pageNumber[_this->TLB.entries-1] = pageNumber;
            _this->TLB.frameNumber[_this->TLB.entries-1] = frameNumber;
        }
    }
```

We need to indicate that new entry has been added, until all frames in TLB have been allocated beforehand.

```
    if(_this->TLB.entries < TLB_SIZE){
        _this->TLB.entries++;
    }
```

Determining Physical Address

After visiting the `find_frame`, we now have enough to calculate the physical address. Then we can print the logical address, physical address, and the value at `physicalMemory[frame_number][offset]`

```
# getPage( virtual_memory * _this, int logical_address)
    physical_address = (frame_number << 8) | offset;
```

Statistics

After completion, the program will report stats on:

1. Page-Fault rate
2. TLB hit rate

```
void print_stats(virtual_memory * _this, double total_addresses){

    // calculate and print out the stats
    printf("Number of translated addresses = %.0f\n", total_addresses);

    printf("Page Miss Rate: %.3f\n",_this->pager.faults / total_addresses);
    printf("TLB Hit Rate: %.3f\n", _this->TLB.hits / total_addresses);

}
```

Testing Correctness

To test correctness of this program, it was provided that the correct output is given in `correct.txt` and it was need to generate the correct output by passing a file called `addresses.txt` (containing integer values representing logical addresses ([0, 65535])) to the program. Results can be seen next section.

Results

Output

Example output of program:

```
# ...
Virtual address: 8940 Physical address: 46572 Value: 0
Virtual address: 9929 Physical address: 44745 Value: 0
Virtual address: 45563 Physical address: 46075 Value: 126
Virtual address: 12107 Physical address: 2635 Value: -46
Number of translated addresses = 1000
Page Miss Rate: 0.244
TLB Hit Rate: 0.055
```

Difference between the `correct.txt` and the output of the program:

```
1001,1003d1000
< Number of translated addresses = 1000
< Page Miss Rate: 0.244
< TLB Hit Rate: 0.055
```

Summary

I think this was a great program. It was interesting dealing with the challenges of creating an virtual memory manager. This program has thought me a lot of about paging, physical memory, translation look-aside buffer, address management. I had a hard time wrapping my head around how the TLB works and finding if the frame number existed.

Appendix