Національний технічний університет України
“Київський політехнічний інститут ім. Ігоря Сікорського”

Факультет прикладної математики
Кафедра системного програмування і спеціалізованих комп’ютерних систем

ЛАБОРАТОРНА РОБОТА №2

з дисципліни

“Архітектура комп’ютерів ”

ТЕМА: “Перетворення віртуальних адрес”

Група: КВ-13

Виконав: Луценко Б. А.

Оцінка:

Київ – 2024

Мета роботи

• ознайомитись із елементами рівня архітектури системи команд;
• ознайомитись з елементами рівня архітектури операційної системи на прикладі функції реалізації і підтримки віртуальної пам’яті; навчитись перетворювати віртуальні адреси у фізичні.

Постановка задачі

Завдання лабораторної роботи наступне: реалізувати програму мовою C або C++, що виконує зчитування послідовності команд (програми) з файлу і заміняє віртуальні адреси на фізичні в командах, що визначаються варіантом. Тип організації пам’яті також визначається варіантом. Заміна адреси відбувається у випадку, якщо сторінка та/або сегмент знаходиться в оперативній пам’яті (ОП). Якщо потрібна віртуальна сторінка та/або сегмент відсутні в ОП, тоді має бути виведене повідомлення про помилку відсутності сторінки/сегменту, й аналіз команд має бути продовжено. Таблиця сторінок/сегментів задається у файлі формату CSV.

Завдання за варіантом

12	сегментно-сторінкова	РС: 2 Кбайт, РТД: 2048	1, 2, 3, 4, 5, 24, 25, 26, 28

№	Команда	Код команди (0x)	Опис
1	MOV <reg1>, <reg2>	1A /reg1 /reg2	перемістити значення з регістру <reg1> у регістр <reg2>
2	MOV <reg>, <addr>	1B 0 /reg /addr	перемістити значення з ОП за адресою <addr> у регістр <reg>
3	MOV <addr>, <reg>	1B 1 /reg /addr	перемістити значення з регістру <reg> в ОП за адресою <addr>
4	ADD <reg1>, <reg2>	01 /reg1 /reg2	додавання значення з регістру <reg1> до значення з регістру <reg2> і збереження результату в регістрі <reg1>
5	ADD <reg>, <addr>	02 0 /reg /addr	додавання значення з регістру <reg> до 4-байтового значення з ОП за адресою <addr> і збереження результату в регістрі <reg>
24	JG <shift>	94 /shift	перехід за 1-байтовим відносним зміщенням <shift> у випадку, якщо ZF = 0 і SF = OF
25	JG <addr>	95 /addr	перехід за 4-байтовою адресою <addr> у випадку, якщо ZF = 0 і SF = OF
26	CMP <reg1>, <reg2>	80 /reg1 /reg2	порівняння двох значень і встановлення відповідних прапорців
28	MOV <reg>, <lit16>	1C 1 /reg /lit16	переміщення 2-байтового числа у регістра <reg>

address.py

```
import src.config as cfg
from src.exceptions import PageNotExistsInROMError, PageNotExistsInRAMError,
SegmentNotExistsInDescriptorTableError, InvalidMemoryAddressError

import csv

# segments descriptors table contains the following fields:
# 1. segment number
# 2. segment pages table path
# 3. pages count
segments_descriptors_table = []

# segment pages table contains the following fields:
# 1. page number
# 2. existence bit
# 3. frame number
segment_pages_table = []

def load_segments_table():
    with open(cfg.SEGMENTS_TABLE_FILE, "r") as f:
        reader = csv.reader(f)
        for row in reader:
            segments_descriptors_table.append(row)

def load_pages_table(segment_number):
    try:
        with open(cfg.DATA_DIR + segments_descriptors_table[segment_number][1] +
".csv", "r") as f:
            reader = csv.reader(f)
            for row in reader:
                segment_pages_table.append(row)
    except IndexError:
        raise SegmentNotExistsInDescriptorTableError(segment_number)

def form_phys_addr(virtual_addr: str) -> str:
    """
    Translates a virtual address to a physical address.

    The virtual address is a 32-bit value, divided into three parts:
    - The first 11 bits (0-10) represent the page offset.
    - The next 10 bits (11-20) represent the page number.
    - The final 11 bits (21-31) represent the segment number.

    The translation process is as follows:
```

```
    1. Retrieve the segment descriptor using the segment number.
    2. Retrieve the page table from the segment descriptor.
    3. Retrieve the frame number from the page table.
    4. Concatenate the frame number with the page offset to form the physical address.
    5. Return the physical address.

    Args:
        virtual_address (int): The virtual address to be translated.

    Returns:
        int: The translated physical address.
    """

    # check if the address is valid
    if len(virtual_addr) != 8:
        raise InvalidMemoryAddressError(virtual_addr)

    try:
        binary = f'{int(virtual_addr, 16):032b}'
    except ValueError:
        raise InvalidMemoryAddressError(virtual_addr)

    page_offset = int(binary[:11], 2)
    page_number = int(binary[11:21], 2)
    segment_number = int(binary[21:], 2)

    load_segments_table()

    load_pages_table(segment_number)

    # check if the page is loaded
    try:
        if segment_pages_table[page_number][1] == '0':
            raise PageNotExistsInRAMError(virtual_addr)
    except IndexError:
        raise PageNotExistsInROMError(virtual_addr)

    frame_number = int(segment_pages_table[page_number][2], 2)

    return f'{int(f"{frame_number:021b}{page_offset:011b}", 2):04X}'
```

---

analyzer.py

```
import src.config as cfg
from src.address import form_phys_addr
from src.exceptions import InvalidMemoryAddressError, PageNotExistsInRAMError,
SegmentNotExistsInDescriptorTableError, PageNotExistsInROMError
```

```
def form_addr(tokens: list[str]) -> str:
    virtual_addr = ''
    for token in tokens:
        virtual_addr += token
    return f'[{form_phys_addr(virtual_addr)}]'

def clean_data(data: str) -> str:
    return "".join(data.split())

def get_tokens(data: str) -> list[str]:
    return [data[i:i+2] for i in range(0, len(data), 2)]

def print_row(tokens: list[str], mnemonic: str, operands: int, error: str = None,
file=None):
    print(' '.join(tokens), file=file)
    if error is not None:
        print(f'{error}', file=file)
    print(f'{mnemonic} {", ".join(operands)}', file=file); print(file=file)

def analyze(input_file, output_dir):
    print("Analyzing file:", input_file)
    print("Output directory:", output_dir)

    CLEANED_FILE = output_dir + "program_cleaned.txt"
    ANALYSIS_FILE = output_dir + "analysis.txt"

    # read the input file and clean the data from whitespaces
    with open(input_file, "r") as f:
        data = f.read()
        data = clean_data(data)

    # write cleaned data to program_cleaned.txt
    output_file = CLEANED_FILE
    with open(output_file, "w") as f:
        f.write(data)

    # read the cleaned data assuming 2 characters as a single token - byte
    with open(output_file, "r") as f:
        data = f.read()
        tokens = get_tokens(data)

    with open(ANALYSIS_FILE, "w") as f:
        # analyze the tokens
        i = 0
        command = []
        current_token = ''
        arguments = []
```

```
        reverse_flag = False
        while i < len(tokens):
            error = None
            current_token = tokens[i]

            if current_token in cfg.OPCODES:
                opcode = current_token
                i += 1
                # i: index of the first operand
                command = [opcode]

                if opcode in cfg.REVERSE_OPCODE:
                    reverse_flag = tokens[i][0] == '0'

                operands = cfg.OPCODES[opcode]['operands']

                if len(operands) == 1:
                    operand_size = cfg.OPERAND_SIZES[operands[0]]

                    if operands[0] == 'SHIFT':
                        arguments = [f'{int(tokens[i], 16)}']
                        command.append(tokens[i])
                    elif operands[0] == 'ADDR':
                        try:
                            arguments = [form_addr(tokens[i:i+operand_size])]
                        except Exception as e:
                            error = str(e)
                            arguments = [f'[0x{"".join(tokens[i:i+operand_size])}]']
                        command.extend(tokens[i:i+operand_size])

                    i += operand_size

                elif len(operands) == 2:
                    operands_sizes = [cfg.OPERAND_SIZES[operand] for operand in
operands]

                    if operands[0] == 'REG' and operands[1] == 'REG':
                        arguments = [f'R{int(tokens[i][1], 16)}',
                                     f'R{int(tokens[i][0], 16)}']
                        command.append(tokens[i])
                        i += 1  # REG REG

                    elif operands[0] == 'REG' and operands[1] == 'ADDR':
                        command.append(tokens[i])
                        try:
                            arguments = [form_addr(tokens[i+1:i+operands_sizes[1]+1]),
                                         f'R{int(tokens[i][1], 16)}']
                        except Exception as e:
```

```
                            error = str(e)
                            arguments =
[f'[0x{"".join(tokens[i+1:i+operands_sizes[1]+1])}]',
                                        f'R{int(tokens[i][1], 16)}']
                        command.extend(tokens[i+1:i+operands_sizes[1]+1])

                        i += 1                  # REG
                        i += operands_sizes[1]  # ADDR

                    elif operands[0] == 'REG' and operands[1] == 'LIT16':
                        arguments = [f'R{int(tokens[i][1], 16)}',
                                     f'{int(tokens[i+1], 16)}']
                        command.append(tokens[i])
                        command.append(tokens[i+1])
                        i += 1 # REG
                        i += 2 # LIT16

                    if reverse_flag:
                        arguments = arguments[::-1]
                        reverse_flag = False

                print_row(command,
                          cfg.OPCODES[opcode]['mnemonic'],
                          arguments,
                          error=error,
                          file=f)


    print("Analysis complete")
```

---

config.py

```
INPUT_DIR = "input/"
OUTPUT_DIR = "output/"
DATA_DIR = "data/"

INPUT_FILE = INPUT_DIR + "program.txt"

OUTPUT_FILE = OUTPUT_DIR + "result.txt"

SEGMENTS_TABLE_FILE = DATA_DIR + "segments.csv"

OPERAND_SIZES = {
    'REG': 1,
    'ADDR': 4,
    'SHIFT': 1,
```

```
    'LIT16': 4
}

OPCODES = {
    '1A': {'mnemonic': 'MOV', 'operands': ['REG', 'REG']},
    '1B': {'mnemonic': 'MOV', 'operands': ['REG', 'ADDR']},
    '01': {'mnemonic': 'ADD', 'operands': ['REG', 'REG']},
    '02': {'mnemonic': 'ADD', 'operands': ['REG', 'ADDR']},
    '94': {'mnemonic': 'JG',  'operands': ['SHIFT']},
    '95': {'mnemonic': 'JG',  'operands': ['ADDR']},
    '80': {'mnemonic': 'CMP', 'operands': ['REG', 'REG']},
    '1C': {'mnemonic': 'MOV', 'operands': ['REG', 'LIT16']},
}

REVERSE_OPCODE = [
    '1B',
    '02',
    '1C'
]

PAGE_SIZE = 2*2**10
DESCRIPTOR_TABLE_SIZE = 2048
```

---

exception.py

```
class CustomError(Exception):
    """Base class for other exceptions"""
    pass

# MEMORY ERRORS

class PageNotExistsInRAMError(CustomError):
    """Raised when the page does not exist in RAM"""
    def __init__(self, virtual_address: str , message="Page does not exist in RAM"):
        self.message = message + f" for address {virtual_address}"
        super().__init__(self.message)

class PageNotExistsInROMError(CustomError):
    """Raised when the page does not exist in memory"""
    def __init__(self, page_number: int, message="Page does not exist in memory"):
        self.message = message + f" for page number {page_number}"
        super().__init__(self.message)

class SegmentNotExistsInDescriptorTableError(CustomError):
    """Raised when the segment does not exist in the descriptor table"""
    def __init__(self, segment_number: int, message="Segment does not exist in the
descriptor table"):
```

```
        self.message = message + f" for segment number {segment_number}"
        super().__init__(self.message)

class InvalidMemoryAddressError(CustomError):
    """Raised when the memory address is invalid"""
    def __init__(self, virtual_address: str, message="Invalid memory address"):
        self.message = message + f" for address {virtual_address}"
        super().__init__(self.message)
```

Тестування програми

program.txt

```
1A 01
1A 9F
1A FF

1B 00 00000001
1B 05 1000FF00
1B 05 00070101
1B 10 00C000A1

1B 15 1000FF00
1B 15 00070101

01 01
01 9F
01 FF

02 00 00C000A1
02 05 1000FF00
02 05 00070101

94 03
94 09
94 0D
94 0F
94 10
94 1F
94 FF

95 00C000A1
95 1000FF00
```

```
95 00070101

80 01
80 9F
80 FF

1C 10 1234
1C 18 1234
1C 1D 1234
1C 1F 1234
1C 1F 0001
1C 1F 0100
```

---

analysis.txt

```
1A 01
MOV R1, R0

1A 9F
MOV R15, R9

1A FF
MOV R15, R15

1B 00 00 00 00 01
MOV R0, [DD565000]

1B 05 10 00 FF 00
Segment does not exist in the descriptor table for segment number
1792
MOV R5, [0x1000FF00]

1B 05 00 07 01 01
Segment does not exist in the descriptor table for segment number 257
MOV R5, [0x00070101]

1B 10 00 C0 00 A1
Segment does not exist in the descriptor table for segment number 161
MOV [0x00C000A1], R0

1B 15 10 00 FF 00
```

```
Segment does not exist in the descriptor table for segment number
1792
MOV [0x1000FF00], R5

1B 15 00 07 01 01
Segment does not exist in the descriptor table for segment number 257
MOV [0x00070101], R5

01 01
ADD R1, R0

01 9F
ADD R15, R9

01 FF
ADD R15, R15

02 00 00 C0 00 A1
Segment does not exist in the descriptor table for segment number 161
ADD R0, [0x00C000A1]

02 05 10 00 FF 00
Segment does not exist in the descriptor table for segment number
1792
ADD R5, [0x1000FF00]

02 05 00 07 01 01
Segment does not exist in the descriptor table for segment number 257
ADD R5, [0x00070101]

94 03
JG 3

94 09
JG 9

94 0D
JG 13

94 0F
JG 15

94 10
JG 16
```

```
94 1F
JG 31

94 FF
JG 255

95 00 C0 00 A1
Segment does not exist in the descriptor table for segment number 161
JG [0x00C000A1]

95 10 00 FF 00
Segment does not exist in the descriptor table for segment number
1792
JG [0x1000FF00]

95 00 07 01 01
Segment does not exist in the descriptor table for segment number 257
JG [0x00070101]

80 01
CMP R1, R0

80 9F
CMP R15, R9

80 FF
CMP R15, R15

1C 10 12
MOV R0, 18

1C 18 12
MOV R8, 18

1C 1D 12
MOV R13, 18

1C 1F 12
MOV R15, 18

1C 1F 00
MOV R15, 0
```

```
1C 1F 01
MOV R15, 1
```